EUUG

European UNIX® systems User Group

UNIX® around the World

10th Anniversary Conference

Conference Proceedings

11th–15th April 1988

LONDON

EUUG

European UNIX® systems User Group

UNIX Around the World

Proceedings of the
Spring 1988 EUUG Conference

April 11-15, 1988
The Queen Elizabeth II Conference Centre
London

ISBN 0 9513181 0 1

Further copies of the proceedings may be obtained from:

EUUG Secretariat
Owles Hall
Buntingford
Herts
SG9 9PL
United Kingdom

ACKNOWLEDGEMENTS

Sponsored by:	European UNIX systems User Group	
Programme Chair:	Sunil K Das	City University London
Programme Committee:	Giuseppi Attardi	Delphi SpA, Italy
	Michel Gien	Chorus Systems, France
	Jim McKie	Bell Communications Research, USA
Proceedings Production:	Jan-Simon Pendry	Imperial College London
	Dave Edmondson	Imperial College London
	Stuart McRobert	Imperial College London
	Diomidis Spinellis	Imperial College London
	Mark Taylor	Imperial College London
Local Organisation:	Mick Farmer	Birkbeck College London
	Zdrav Podolski	Insignia Solutions, UK

The continuous cheerful support from many friends has led to the organisation of the EUUG's three day Conference, one-day tutorials, and the UNIX Showcase. The debt owed to the Programme Committee, Local Organisers and the Proceedings Production Team is immeasurable. We are grateful to many people who have made substantial contributions in other areas and expended much effort "beyond the call of duty." In addition to acknowledging those mentioned above, the following deserve special note: Tom Cargill, Michael Lesk, Doug McIlroy, Gilly McKie, Marion Podolski, Peter Salus, Hazel Tomlinson and Neil Urquhart. Furthermore, the sponsorship or supply of resources from the following organisations is gratefully recorded: City University London, Gould Electronics Ltd and the UK UNIX systems User Group.

Also, we wish to express our appreciation to all those authors who submitted their papers in time for inclusion in the Proceedings. Nearly 60 papers were submitted to the Programme Committee and the selection process proved very difficult. Moreover, we were constrained by the fact that we wanted to accept papers from as many different countries as possible which had supplied extended abstracts of high enough standard. The outcome was that the Committee had to disappoint many authors who had submitted papers of very high quality. Such authors should, if appropriate, submit their papers to the Programme Chairman of the next EUUG Conference in Portugal.

Sunil K Das
Proceedings Editor

UNIX Conferences in Europe 1977–1988

UKUUG/NLUUG meetings

1977 May	Glasgow University, Glasgow
1977 September	University of Salford, Salford
1978 January	Heriot Watt University, Edinburgh
1978 September	Essex University, Colchester
1978 November	Vrije University, Amsterdam
1979 March	University of Kent, Canterbury
1979 October	University of Newcastle
1980 March	Vrije University, Amsterdam
1980 March	Heriot Watt University, Edinburgh
1980 September	University College, London

EUUG Meetings

1981 April	CWI, Amsterdam, The Netherlands
1981 September	Nottingham University, UK
1982 April	CNAM, Paris, France
1982 September	University of Leeds, UK
1983 April	Wissenschaft Zentrum, Bonn, Germany
1983 September	Trinity College, Dublin, Eire
1984 April	University of Nijmegen, The Netherlands
1984 September	University of Cambridge, UK
1985 April	Palais des Congres, Paris, France
1985 September	Bella Center, Copenhagen, Denmark
1986 April	Centro Affari/Centro Congressi, Florence, Italy
1986 September	UMIST, Manchester, UK
1987 May	Helsinki/Stockholm, Finland/Sweden
1987 September	Trinity College, Dublin, Ireland
1988 April	Queen Elizabeth II Conference Centre, London, UK

Technical Programme

UNIX Around the World

Sunil K Das

City University London,
Computer Science Department
London EC1V 0HB, UK
sunil@cs.city.ac.uk

ABSTRACT

In the Preface to the Eighth and Ninth Editions of the Programmer's Manual for the UNIX Time-Sharing System, Doug McIlroy says that the volumes describe the lineal descent of the original operating system pioneered by Ken Thompson and Dennis Ritchie. Distributed computing proved to be the distinctive theme of the landmark Eighth Edition: Dennis Ritchie's coroutine-based stream IO system, and the Datakit virtual circuit switch realisation by Lee McMahon and Bill Marshall, provided the basis for networking, Peter Weinberger's remote file systems made it painless, and Rob Pike's software for the Teletype 5620 moved system action right out to the terminal.

Users distributed around the world is the theme of the Spring 1988 EUUG Conference. The Conference Chairman discusses here why users around the world have demanded to use UNIX, why UNIX has proved successful around the world, and the future of the UNIX system in the world marketplace.

The paper finishes with a citation of the original and innovatory contributions made by many of the speakers who travelled from all over the world to be at the the EUUG's Conference held at the Queen Elizabeth II Conference Centre, London in April 1988.

1. The Worldwide Demand for UNIX

In an AT&T Bell Laboratories Technical Report, Doug McIlroy says that the UNIX system has brought great honour to its primary contributors, Ken Thompson and Dennis M Ritchie, and has reflected respect upon many others who have built on their foundation [McIlroy 1987]. Without the vision of Ken Thompson, UNIX would not have come into existence; without the insight of Dennis Ritchie, it would not have evolved into a polished presence; without the imagination of Michael Lesk, and the popularising touch of Brian Kernighan, it would not have acquired the extroverted personality that commands such widespread loyalty.

Globally, technical and professional users in the software engineering and information technology sectors – drawn from such diverse backgrounds as computer manufacturers, end-users, software houses, universities and research centres – have demanded to use the UNIX system. This privileged community can stand on the shoulders of the intellectual giants who have gone before. By building on the work of others, a generation of software engineers has been spawned who have become practitioners of the philosophy that has characterised and underpinned the UNIX programming environment.

The UNIX system has proved to be a grassroots product. There was no market research and no glossy sales campaign to bring it to the position it enjoys today. From the beginning, UNIX was released to an ever widening circle of users only because those users demanded it. AT&T Bell Laboratories, it appears, was dragged kicking and screaming into providing UNIX to the world.

Ken Thompson and Dennis Ritchie wanted to build something they would enjoy using [Mohr 1985]. They succeeded, and other users want to share this enjoyment. Users realised that UNIX was not Utopia, but it did provide, and still does provide, the best set of software tools around.

The UNIX system provides a clean, flexible interface to the programmer, which is available on various different computers because UNIX is relatively portable. This is a great advantage to the user who wishes to move his skills, programs and data between different computer systems without learning a new command interpreter and system conventions. One is reminded of Fortran's great achievement in the

1950's which provided a convenient-to-use and easily-understood notation usable on different manufacturer's hardware [Holt 1983]. It is tempting to call UNIX the "Fortran of operating systems", destined to become the standard operating system for various application areas!

UNIX was designed to be easily understood, used and modified by software designers. For users and software buyers, UNIX means spending less on software. For end users, UNIX means cheaper, more timely, and more portable software applications, and shorter development times.

Hardware gets cheaper by the month, while software design times have become steadily more expensive. There is greater pressure than ever before for more powerful operating systems and languages. Users know that UNIX and C make good economic sense.

UNIX is in wide use on micros, minis and mainframes and is now even spoken of as an industry standard. Even with its shortcomings, UNIX is an excellent base for personal computers and business applications.

2. The Success of UNIX

There were sociological forces which helped in the success of UNIX [Ritchie 1985]. The 1970s was the decade of the minicomputer of which the most successful range was that of the PDP-11. Until then, timesharing systems had only been developed for big machines. In 1970, Bell Labs' Legal Department needed a word processing system, and a newly announced PDP-11/20 was eventually purchased. The UNIX project needed more computing power, and these two needs complimented each other. By 1971, the first real users of UNIX were three typists entering patent applications. Thus, from the outset, UNIX had to satisfy researcher and ordinary users alike. Unexpectedly, word processing has become the single most commonly used computer application, and the tools for this were provided from the outset.

Portability was on the horizon by 1973; applications could continue to run while the hardware underneath changed. In addition, UNIX enjoyed an unusually long gestation period (10 years) under the control of its designers, with feed back from users (i.e. Universities).

Technically, UNIX is a simple, coherent system which pushes a few good ideas to the limit. The greatest intellectual achievement embedded in UNIX is the success Thompson and Ritchie had in understanding how much you could leave out of an operating system without impairing its capability [Vyssotsky 1985]. Being a modular operating system, UNIX positioned at the command layer, many utilities which had previously been included in the kernel.

Much of the success is due to UNIX being written in a high level language [Kernighan and Pike 1984]. First, UNIX is relatively portable because it is written in C. A strong commercial advantage is that it runs on a range of computers from micros, to minis to mainframes. Second, the source code is available, and is easier to understand than other operating systems because it is written in C. This makes it easy to adapt to particular requirements. Third, :t provides a pleasing work environment, especially for programmers. It's a "good" operating system!

The UNIX System comes with a rich set of software tools; in excess of 300 on some UNIX systems. Moreover, it excels at enabling small programs to be combined with others to do more complex tasks. The UNIX system introduces a number of innovative programs and techniques, but there is neither a single program nor an idea that makes it work well. What makes UNIX effective is an approach to programming, a philosophy of using a computer. The "power" of UNIX comes from the relationships that can be generated between programs, not from the programs themselves. Many UNIX programs do quite trivial tasks in isolation, but combined with other programs they become general purpose and useful software tools [Kernighan and Pike, 1984].

3. The Marketplace

The future for the UNIX system has been addressed before [Das and Farmer 1988]. The commercial acceptance of UNIX as a product is on the rise, in particular with small and medium sized businesses. Market research predicts that UNIX applications will grow from $2.8 billion in 1986 to $10 billion in 1990. Much of the commercial interest in UNIX centres around its portability which allows customers to escape from being locked in to any one supplier's hardware and software. It also allows them to grow from low end micros to Cray super-computers with one operating system.

UNIX is traditionally very strong in certain sectors and is gaining in popularity in others. Almost all workstation suppliers offer some version of UNIX. These include Apollo, DEC and Sun as well as newcomers such as Apple, IBM and TeleVideo. Emerging graphics standards, such as X-window and PHIGS (Programmer's Hierarchical Interactive Graphics Standard), are already available on UNIX systems. Over the past three years, minisupercomputers have carved out a fast-growing market niche.

UNIX is highly visible in this area which means it is a viable alternative to VMS in the scientific computing market, with C becoming more widely used.

The favourable qualities of UNIX are that it contains multi-tasking, networking, a mature development environment, vendor independence, and over 250,000 trained programmers in the USA alone - more than any other operating system can claim. Conversely, there are too many flavours of UNIX, and currently UNIX is poor on graphics, windowing, and end-user applications. Some feel it has an unfriendly user interface for the non-programmer.

Standardisation will prove important in the future. The draft ''Portable Operating System Interface for Computer Environments'' (POSIX), represents the culmination of several years of intensive effort to develop a standard operating system interface to support the portability of applications software at the C language source code level, and is designed to be used by both software application developers and system implementors. POSIX will provide further, the basis of an evolving environment for new standards which specify the interface bindings for functional areas such as graphics, data bases, communications, file systems, user interfaces, and other languages.

POSIX is expected to be adopted by the IEEE as a full use standard in March, 1988 and has been approved for consideration as an ISO standard. The National Bureau of Standards in the USA intends to adopt POSIX as a Federal Information Processing Standard (FIPS) and therefore is developing a suite of software which will test conformance of an operating system environment to the POSIX FIPS.

4. The EUUG Conference Speakers

The Conference, entitled ''UNIX around the World'', has attracted the very highest quality technical papers, thus affording an international forum for the presentation of current work on a wide variety of topics related to the UNIX and C programming environments. The global nature of the Conference stresses the importance of standards, portability, security and communications. Technical presentations concerning standards like SVID, X/OPEN, POSIX and ANSI C were submitted to the Programme Committee, as well as talks on Secure UNIX, UNIX Networking and real-time UNIX.

Sunil K Das (UK), City University London.

The Conference Programme Chairman and Chairman of the UKUUG has borrowed freely in the descriptions below, thanks in particular go to Doug McIlroy [McIlroy, 1987]. Any factual inaccuracies or misleading statements are unintentional, but the Chairman acknowledges complete responsibility. He is particularly pleased that Dennis M Ritchie and Stephen R Bourne will be attending the Conference.

An impressive array of present and past members of AT&T Bell Laboratories' Computer Science Research Centre (CSRC) will be participating in and/or speaking at the Conference. All of these world famous researchers made original and innovative contributions to the UNIX programming environment. Without any one of the people whose contributions are cited here, neither UNIX nor the work of others in the CSRC would be the same.

Dennis M Ritchie (USA), AT&T Bell Laboratories.

Ken Thompson began the construction of the UNIX system from the ground up based on a file system model worked out with Dennis Ritchie and Rudd H Canaday. Dennis, best known as the father of C, joined Ken very early on. He contributed notions such as *fork-exec* and *set-userid* programs. Jointly, they wrote the *fc* compiler for Fortran IV. The first debugger *db* and the definitive *ed* were Dennis', as was the radically new stream basis for IO in v8, and much networking software. Datakit and streams made possible Peter Weinberger's network file system, and Dave Presotto's connections to diverse networks. As a result the research machine is no longer identifiable; users can - and do - work on one or more of two dozen computers simultaneously. With Steve Johnson, Dennis made UNIX portable, moving the system to an Interdata machine (v7).

Stephen R Bourne (UK), Digital Equipment Corporation.

Steve arrived at the time of v6, bringing Algol 68 with him. His definitive programs, the debugger *adb*, and the ''Bourne shell'' *sh*, although written in C, looked like Algol 68: Steve wrote BEGIN and END, and DO and OD instead of { and }. The Bourne shell almost overnight drove out the simple old shell. A PWB shell had made programming useful; the Bourne shell made it an essential part of UNIX programming (v7). Steve also contributed macro constructs to the UNIX Circuit Design System.

Stephen C Johnson (USA), Ardent Computer Corporation.

Witn *yacc* Steve reduced to practice Al Aho's expertise in language theory (v3). Upon that base, he built the portable C compiler *pcc* (v7) that was used to port the UNIX system, and to evaluate candidate instruction sets for unbuilt machines. *Yacc*, abetted by Mike Lesk's *lex* (v7), stimulated an entire language industry. *Yacc*, by eliminating much drudgery of compiler-writing, made possible the extensive experimentation that underlies many novel languages. Steve also made the first *spell*, worked on computer algebra, and devised languages for VLSI layout.

Michael E Lesk (USA), Bell Communications Research.

With a prescient market instinct, Michael made text formatting accessible to the masses with the generic macros *-ms*, which were to *troff* what a compiler is to assembly language. He rounded out *-ms* with the preprocessors *tbl* for typesetting tables and *refer* for bibliographies. He also made the *lex* generator for lexical analysers. Eager to distribute his software quickly and painlessly, Michael invented *uucp*, thereby begetting a whole global network. *Uucp* gave operational meaning to the phrase "UNIX community" (v7). News now travels electronically among users all over the world; and technical collaborations proceed between distant locations almost as easily as within one building. Over the years, often helped by Ruby Jane Elliott, he initiated fascinating on-line audio, textual, and graphical access to phone books, news wire *apnews* (v8), and *weather* (v8). With Brian Kernighan, he was responsible for the UNIX computer-assisted instruction software, *learn* (v7).

John R Mashey (USA), Mips Computer Systems.

Well known for his hardware, software and CPU design skills, John was a co-author of the BSTJ paper "The Programmers Workbench". During the 10 year period spent at AT&T Bell Laboratories, he worked on command languages, text processing, computer-centre UNIX issues and various features of v7, followed by management of applications projects and exploratory projects with bitmapped displays and programming environments.

M Douglas McIlroy (USA), AT&T Bell Laboratories.

Author of the papers "The UNIX Success Story" and "A Research UNIX Reader", Doug exercised the right of a department head to muscle in on the original two-user PDP-7 system. *Echo*, seemingly the simplest of utilities, originated with Multics, where it was used to test the sanity of the shell. The present version arose as a finger exercise for Doug in C programming (v2). Then it turned out to be useful, a mainstay of shell scripts. His observations and interest in *echo* led to his parable "The UNIX and the Echo" [Kernighan and Pike, 1984], which goes to show that research computer scientists are not so ivory towered that we don't peek at Greek mythology from time to time!

The basic redirectability of input-output made it easy to put pipes in when Doug finally persuaded Ken Thompson to do it. In one feverish night Ken wrote and installed the pipe system call, added pipes to the shell, and modified several utilities, such as *pr* and *ov*, to be usable filters. The next day saw an orgy of one-liners as everybody in the CSRC joined in the excitement of plumbing. Pipes ultimately affected the CSRC's outlook on program design far more profoundly than had the original idea of redirectable standard input and output.

Later Doug contributed an eclectic bag of utilities: *tr* (v4) which was deliberately designed to follow the stream transformational model, *tmg* for compiler writing, based on Bob McClure's compiler-compiler, which originally supplemented the assembler on the tiny PDP-7, *speak* for reading text aloud, *diff*, and *join*. He also collected dictionaries and made tools to use them: *look* (v7), *dict* (v8), and *spell* (v7).

Robert Morris (USA), National Computer Security Centre.

Bob stepped in whenever mathematics was involved, whether it was numerical analysis or number theory. Bob invented the distinctively original utilities *typo*, and *dc–bc* (with Lorinda Cherry), wrote most of the mathematics library, and wrote *primes* and *factor* (with Ken Thompson).

His series of *crypt* programs fostered the CSRC's continuing interest in cryptography. Bob's first file encrypter appeared in v3 with the explicit intent to stimulate code breaking experiments. Stimulate it did! Bob himself broke *crypt* by hand. Later Dennis Ritchie automated the cryptanalysis using methods of Jim Reeds (Berkeley). Complete with an editor interface, a new *crypt* went public in v7. It also succumbed to an attack by Jim Reeds and Peter Weinberger – and fortunately, too: more than one person who locked data in *crypt* and threw away the key was rescued by code breakers.

But the still arduous process of code-breaking is not the easiest way to attack *crypt*. A simpler gambit is to catch a system administrator off guard and install a *Trojan horse* in *crypt* itself to snatch every new secret. Thus the very presence of *crypt* may have just the opposite effect on security from what was intended.

David L Presotto (USA), AT&T Bell Laboratories.

Dave tamed networks! His *upas* brought some order to a Babel of mail addresses and his *ipc* primitives provided a common basis for communication and remote file access via Internet, Ethernet, and Datakit.

Electronic mail was in the UNIX System from the start. Never satisfied with its exact behaviour, everybody touched it at one time or another: to assure the safety of simultaneous access, to improve privacy, to survive crashes, to exploit *uucp*, to screen out foreign freeloaders, or whatever. Not until v7 was the interfaced changed (by Ken Thompson). Later, as mail became global in its reach, Dave Presotto took charge and brought order to communications with a grab-bag of external networks (v8).

Despite the turbulent evolution of mail, to this day a simple postmark is all that it adds to what you write. Old UNIX hands groan at the monstrous headers that come from latter-day mailers and at the fatness of their manuals.

Andrew G Hume (Australia), AT&T Bell Laboratories.

Andrew wrote *proof* to put troff on your screen (v8), parts of the UNIX Circuit Design System (v9), *mk* to supplant *make*, and a remote backup service (v9). His most recent research has been to speed up the grep family of programs, as will be discussed in his paper "Grep Wars".

With a Conference title "UNIX around the World", the Conference Chairman sought internationally known speakers from all over the globe. He has found in the people cited here, innovative researchers who have contributed novel software to the UNIX environment created by Ken Thompson and Dennis Ritchie, and others.

Maurice J Bach (Israel), IBM Haifa Scientific Centre.

While with AT&T Maury wrote the much praised book "The Design of the UNIX Operating System". This unique, authoritative text documents the internal algorithms and structures which form the basis of the kernel, and analyses their relationship to the programmer interface. Much of the material was based upon courses Maury taught for AT&T.

Robert Gingell (USA), Sun Microsystems.

Rob is Manager of New Systems at Sun Microsystems. His areas of interest include operating systems and network architecture, and programming environments. Prior to coming to Sun, he was at Case Western Reserve University from which he received a B.S. in Computer Engineering.

Samuel J Leffler (USA), PIXAR.

Sam is employed by Pixar to carry out research and development in computer graphics. He is currently working on tools for the development of graphics algorithms that operate in parallel processing environments. Previous work focused on the use of computers in 3-D animation and modeling. Recently he has been grappling to produce a public domain version of Sun's NeWS system which means he's one of those rare breeds who has a fair amount of experience with PostScript. Prior to joining Pixar, he was employed at Lucasfilm, and before that was responsible for the 4.2 release of Berkeley UNIX for the VAX.

John Lions (Australia), University of New South Wales.

When John received his 5th Edition UNIX system, UNIX was a noun and was not a trademark! The original licence was dated 15th December, 1974 and the arrival of the tape/manuals proved to be a timely Christmas present. By July 1976, John had published "A Commentary on the UNIX Operating System" and formatted the accompanying "V6 PDP-11/40 Source Code Book". He said that *nroff* yielded some of its more enigmatic secrets so reluctantly, his gratitude was indeed mixed. However, without John's book, many of us would not remember that UNIX was once less than 10,000 lines of code and transportable in a student's brief case. And perhaps that immortal comment on line 2238 might have escaped us:

```
/* You are not expected to understand this */
```

Jun Murai (Japan), University of Tokyo.

Architect and designer of the Japanese UNIX Network (JUNET) which uses *tcp/ip* protocols over dial-up telephone lines, Jun has a hard time explaining that his name really is Jun and JUNET is not named after him!

David Turner (UK), University of Kent.

David is well known for his research into the design and implementation of functional programming languages. He has designed a series of functional programming systems, all running under UNIX, of which "Miranda" is the most recent. He invented a new implementation technique for functional languages based on compilation to combinatory logic. He holds the Chair in Computation at the University of Kent at Canterbury.

5. References

Das, S K and Farmer, M "UNIX and the Future", *Computer Bulletin*, March 1988.

Holt, R C *"Concurrent Euclid, the UNIX System and Tunis"*, Addison-Wesley, 1983.

Kernighan, B W and Pike, R *"The UNIX Programming Environment"*, Prentice-Hall, 1984.

McIlroy, M D "A Research UNIX Reader", AT&T Bell Laboratories, *Computer Science Technical Report No. 139*, 1987.

Mohr, A "The Genesis Story", *UNIX Review*, January 1985.

Ritchie, D M "Reflections on Software Research", *UNIX Review*, January 1985.

Vyssotsky, V "Putting UNIX into Perspective", *UNIX Review*, January 1985.

UNIX Past, Present, and Future: Changing Roles, Changing Technologies

John R. Mashey

MIPS Computer Systems
Sunnyvale, CA 94086

1. Introduction

The UNIX operating system seems to defy the laws of physics by remaining in perpetual motion. This paper takes a brief look at where it's been, where it is, and where it might be going. In particular, UNIX stands as a major beneficiary of the the current developments in RISC microprocessors.

2. History

For years I'd used used the following model of UNIX history:

Roots (Pre-1969)
Birth (1969-1973)
Childhood (1973-1977)
Adolescence (1977-1981)
Maturity (1981-)

Lately, I've added the following:

Maturity (1981-1985)
Second Adolescence (1985-1989), or Here We Go Again

Of course, this is an over-simplification, as anyone would know who's ever seen UNIX version history charts, even partial ones. However, it still seems to offer a useful model of events. I don't know why the eras all ended up 4 years long; perhaps there is a natural periodicity to such things. Important events (new UNIX versions or major technology changes) tended to happen near the end of each period, then spread rapidly. This lets us choose the year just after each boundary (1974, 1978, etc) as an interesting sample point, although the particular choice needs explanation.

Let us look at the process by which new technology spreads. In step 1, some new technology is developed in some research lab or by some other small group of people. For a few years, it may be that nothing is even published about it. In step 2, other groups (usually called "early adopters" in marketing parlance) begin applying the new technology. This may take years, during which the originators and early adopters run about trying to sell the ideas to others, and are somewhat successful, but are often ignored or classed as lunatics. ("You call these minicomputers computers? You can't be serious!") In step 3, exponential growth takes over, and many more people get on board. By step 4, even the most conservative people are finding uses for it, and it becomes an expected part of the environment. The sample years chosen were ones where some technology was in middle-to-late step 2. This is an interesting phase: new technology is strong enough that you know it is successful (unlike things that never make it past step 1), but it can be expected to grow much further.

This model leans more towards the programming and technical uses of UNIX, and also reflects my own biases and experiences.

2.1. 1974

UNIX users were mostly programmers working on UNIX-based projects, mostly in small, tightly related groups, using the recently-available UNIX V5. Most people were using, or moving to the DEC PDP 11/45, about .5 Mips, with a maximum of 248KBytes of memory. A typical configuration cost about $250K. With a minimum (typical) maximum user load of 1(16)24 simultaneous users, each user's average share of CPU and memory was .50(.03).02 Mips, and 248KB(25KB)10KB of memory. People used 300-Baud hardcopy terminals. Support of 24 simultaneous users was accomplished by heroic system administration coupled with the the patience of users desperate to accomplish at least a little work. Nevertheless, such systems were both useful and cost-effective in comparison with mainframe time-sharing, and hence were attracting interest. Addition of space-consuming features to any important program was viewed with extreme suspicion. Acquisition of a new "release" of software was accomplished by driving to Murray Hill to see what new things could be found on the research machine.

2.2. 1978

Many more users were programmers and others that might be using UNIX as a general utility, or to support non-UNIX programming projects. UNIX was UNIX V6, or the USG or PWB versions thereof. People most often used a PDP 11/70 (.8Mips, 1MB memory), with 1(32)48 simultaneous users. This gave each user .80(.03).02 Mips and 1024KB(32KB)21KB. Although computing power had not increased tremendously, the larger memory was quite helpful, since programs shared space efficiently, and both kernel and user processes remained squeezed into 64KB text plus 64KB data. Adding features had certainly become thinkable. People continued to use 300-Baud hardcopy terminals, but more terminals were running at 1200-Baud, and 24X80 CRTs were becoming widespread, sometimes running at higher speeds. A useful system could still be bought for about $250,000, a cost that happened to be low enough that a Bell Labs Director could sign for it with minimal bureaucracy. The result was a proliferation of systems bought by organisations voting with their feet. At least somewhat in response to this trend, PDP 11/70s had begun to appear inside most computer centers.

An important milestone had been reached: porting of UNIX to another hardware architecture.

2.3. 1982

The user community now included many non-programmers, and they most likely used a 4-8MB DEC VAX 11/780, so that 1(32)48 people received 1.0(.03).02 Mips and 4096KB (130KB) 90KB apiece. However, although they *still* hadn't gained much CPU power, many more were using CRTs at speeds high enough to allow routine use of screen editors, and once again, hardware design helped software by letting us throw more memory at the problem. UNIX variants abounded, including 4.1BSD, System III, USG 4.0 (inside BTL), and various V7 offshoots. Many people inside Bell Labs had learned portability lessons from painful experiences of porting PDP-11 code to other machines. UNIX kernels and user processes were permitted to grow beyond 64KB text plus 64KB data, thus removing the major technical barrier to creeping featurism. People were busy porting UNIX V7 to microprocessors, and the IBM PC had Happened, although most true UNIXers looked down upon the primitive facilities available thereon. Local networks were beginning to be used, albeit haphazardly, and people were looking at bitmapped displays for UNIX.

2.4. 1986 (NOW – really 1988)

1986

It is much more difficult to paint a dominant mode of usage, even in the technical area alone. People may still share a mainframe, mini-super, or large supermini. For example, 1(32)80 might share a 20MB, 4.2 Mips VAX 8600, so that each receives 4.2(.13).05 Mips, and 20MB(.7MB).25MB. A super-micro may be be shared by a smaller number of people, so that a 2 Mips CPU with 4MB memory is shared by 1(8)16 people, giving each 2(.25).12 Mips, and 4MB(.5MB).25MB of memory. Finally, single-user UNIX machines are much more common, ranging from PCs to technical workstations, which give the individual user 1-2 Mips and 4MB or more of memory. Much computing is now done in heterogeneous networks. Options and features have proliferated, but the improved user interfaces offered by higher-bandwidth displays help mitigate this effect upon the user. Companies tolerating, using, or owing their existence to UNIX have also proliferated. UNIX versions have consolidated around System V Release 2 or 4.2BSD, with SVR3 and 4.3BSD beginning to appear.

1988 (REALLY NOW)

What's changed mostly is the availability of 3-4-mips PCs and workstations. Pointing the way towards 1990, inexpensive, but high-performance (10 mips) VLSI RISC microprocessor systems started appearing in 1987, and 20-mips ones are imminent.

UNIX versions have been updated to 4.3BSD and SVR3, and many combined versions have appeared, are appearing, or are being promised to appear.

2.5. 1990

We can safely assume that the computing environment will have diversified even more than it has already. Some groups of people will always seek minimal-cost solutions, and will therefore either use low-cost PCs with file servers or divide a larger computer. Presumably applications will grow, so that most people will get (and need!) at least .5Mips and .5MB apiece. Anyone who needs more power will be able to buy a 10-20 Mips, 16-64MB single-user workstation for entry costs of less than $1,000 / Mips, and such things will be in widespread use, meaning that they better have started to appear by 1988. [1988 note: 10-mips ones started shipping in late 1987.] Small servers (i.e., still not necessarily in the computer room) will offer 30-50 Mips and 64MB-512MB, at costs of $1,000-$2,000 / Mips. High-end single-user workstations should offer similar performance. Mini-supers will have gotten more super.

Almost anything that is not currently networked will be, ranging from inexpensive low-speed nets for the office to high-speed fibre-optic nets among bigger workstations and servers. Almost every new computer architecture will run UNIX. However, kernel underpinnings may have changed to ones like V (Stanford) or Mach (CMU) that are more directly aimed at distributed systems. There will still be no standard UNIX, although many will have incorporated SVID, X/Open, and POSIX. As always, there will be a horde of UNIX variants that differ in those areas nearest the state of the art. Thus, there will probably be several competing standards for graphics and especially windowing, although that area should have improved from today's current disaster area. (X11 and/or X11/NeWS at least offer some hope.) Networking should improve in a similar fashion. There will probably be hordes of competing representations for complex documents, image, and voice.

Note that the speed and cost predictions are *conservative* ones, based on reasonable technology trend analyses. However, applications will have grown to consume this power, just the way they jumped from 11/70s to VAXen, and immediately gobbled everything. It is safe to predict that some people will expect much more. VLSI designers will still want CRAYs on their desks.

Now, let's go back and look at different areas in which UNIX has evolved. In each section I offer a summary, setting 1974 UNIX to 5, then rating the rest of computing and later UNIX systems on that scale. Bigger numbers are better; the scale is arbitrary; all opinions are my own; they're mainly meant to stir discussion on where we might go.

3. Individual Programming

The earliest place to look in UNIX for leverage is for the individual programmer, i.e., in aspects that improve an individual's productivity.

3.1. Editors, Interfaces, and Data

Here we cover the fundamental leverage area of the energy required to make the machine do something, specifically in terms of the immediate human interface [editors, command languages] and file system.

Leverage Summary

1974		1978		1982		1986		1990	
UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest
3-5	0-6	5-7	0-8	5-8	0-10	8-12	0-15	8-20	0-20

Even early UNIX offered the programmer working leverage by simply suppressing unwanted details of disk allocation and access, and by offering simple, inexpensive, and relatively powerful human interfaces.

In 1974, the typical UNIX programmer used a 30-character-per-second hardcopy terminal, unendurable by current standards. However, 300 baud, *ed*, and *sh* were often thought wonderful <5> when compared with the alternatives. Those who have never had the experience of allocating tracks of disk space via IBM JCL cards <0> might try this once to appreciate just what a wonderful improvement UNIX was at that point. Many programmers still submitted decks of punched cards for batch processing, or used expensive, restricted line editors that lacked even regular expression pattern matching. In fact, many were pleased to be able to edit such decks as UNIX files and run them via Remote Job Entry <3>.

By 1982, most people expected to use CRT terminals running at 1200 baud, Screen editors, such as *vi* or *emacs* became widely available<8>, and remain so today. Shells also improved, offering (early) more programmability and (later) more powerful terminal interfaces.

Although the above mode of interaction remains dominant <8>, use of bitmap displays as terminals or workstations has now become more common <10-12>, with noticeable improvements starting around 1983-1984. Such use clearly changes the potential modes of interaction to include much heavier use of display bandwidth, pointing devices, and graphics.

UNIX has certainly been a better host to advanced interfaces than have many older operating systems: many technical workstations are based on UNIX. On the other hand, UNIX is still plagued by a lack of standards for graphics window-management, although the widespread support for X-Windows promises some hope here. Finally, compared to workers in Computer-Aided Design, software engineers are woefully under-served.

To summarise, for many years UNIX had one of the most convenient interfaces, when compared with mainframe and other minicomputer systems. For a while, others have surpassed it in some areas by running on personal computers too small for UNIX. As microprocessor system costs have dropped, UNIX has become available on single-user workstations, and is starting to catch up in immediate interfaces. Hopefully, by 1990, it will have interfaces as good as the best of the rest, and perhaps advantages over PC operating systems that have been extended beyond their natural design points.

3.2. Text Processing

''Why can't be more like my Mac, and vice-versa.''

This has long been a strength of UNIX, but has, unfortunately, not progressed as fast as one might like. This area is quite important to programmers and most other users. In the early life of the Programmer's Workbench, it was often noted that people justified their usage by predicting programming productivity, but what they really did was documentation.

Leverage Summary

1974		1978		1982		1986		1990	
UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest
5	0-4	8	0-6	8-10	2-12	8-12	2-20	8-30	2-30

In 1974, what most people got was raw *nroff*<5>, although a few lucky groups had phototypesetters and access to *troff*. By 1978, many people used *tbl*, *eqn*, and powerful formatting macro packages <8>. Larger typesetters became widely available through computer centers. In general, the UNIX toolkit was as good, and usually better, than other widely-available systems. Some new tools have been added since then <10>.

By 1986, inexpensive laser printers are widely available. By 1988, they're *cheap*.

This area has supported continual engineering improvements, but has seen few real breakthroughs. For example, *troff* externals have changed little in 10 years, and several of the still-popular macro packages (–MS, –MM, for example) were written 10-12 years ago. Ease-of-use of *troff* and related programs is nowhere near that of some of the current desktop publishing systems <20>, although *troff* and its friends can still do some things the others cannot. The others are catching up fast. At least UNIX is also host to systems like Interleaf or Frame. An ideal system <30> would combine the convenience and ease of use of these with the expressive power and control of the *troff* complex, and research versions of this nature are starting to be seen. Unfortunately, the ideal is nowhere near in heavy use yet, and it probably awaits the more powerful workstations alluded to above, i.e., 10 Mips on the desktop. I do have hopes that the desktop publishing system I've always wanted will be available by 1990 on UNIX if only because more systems are being written in C.

3.3. Programming Languages and Level of Work

This is intended to measure the amount of work it takes to write new programs, debug them, and maintain them. Since there exist many problem domains and languages to handle them, this section is of necessity an over-simplification focused on systems programming and related activities.

Leverage Summary

1974		1978		1982		1986		1990	
UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest
2-5	0-4	5-10	0-12	7-12	0-15	8-15	0-20	8-30	0-25

In 1974, being able to use C instead of assembler for systems programming was wonderful <5>.

By 1978, shells had long since become programmable, other useful tools, like *make*, *awk*, and SCCS were in use <10>.

During 1977-1981, I remember a great deal of experimentation and creation of special-purpose languages, but somehow, it didn't seem like there was that much clearly-observable widespread progress in this area <12>.

By 1986, many more special-purpose languages exist, including the application generators and 4GLs that now routinely accompany database systems. Also, object-oriented ideas have now infiltrated UNIX, by way of C++, or Objective-C, for example <15>. However, many people still write in the same C that existed in 1978.

In 1990, one can safely assume that raw C (in the form of ANSI standard C) will still be popular, although languages like C++ should be in heavy use. I'd hope that we'll be able to move to more highly-leveraged languages (SMALLTALK, LISP, PROLOG), constraint-based systems (like van Wyk's *ideal*), better debuggers, and other tools that burn Mips to provide expressiveness <25>. The soon-to-occur jump in performance and cost/performance should make this possible.

4. Group Programming

These attributes measure programming as a group activity. Note that minicomputer-based operating systems have often done better than either mainframes or personal computers in this area.

4.1. Shared Data

This attribute measures the ease of sharing data.

Leverage Summary

1974		1978		1982		1986		1990	
UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest
5	0-6	7	0-8	10-12	0-12	10-15	0-16	10-25	5-25

In 1974, UNIX was much stronger than most systems in offering convenient sharing of files <5>.

By 1978, some modest improvements had been made in areas of protection and ease-of-use when multiple groups shared machines <7>.

By 1982, various groups had implemented networked filesystems of various ilks; that they existed was good; that they remained of various ilks was bad <10-12>. Various vendors (Apollo, Prime, DEC, for example) had implemented some good homogeneous networked filesystems.

By 1986, there exist several vendor-supported proprietary UNIX-based network filesystems. AT&T's RFS has been released, and Sun's NFS is widely used <10-15>.

By 1990, most UNIX systems will support some network file system, probably either NFS or RFS, or both. Performance improvements in both CPUs and networks should help this process be more cost-effective.

4.2. Shared Environments

Sharing work means more than sharing data conveniently; it includes the use of maintenance tools (like SCCS), communications tools (like E-mail), and any other meant to help programming as a group activity. This attribute measures the efficiency of this activity.

Leverage Summary

1974		1978		1982		1986		1990	
UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest
5	0-5	5-10	0-10	5-15	0-15	5-20	0-20	10-30	0-30

In 1974, UNIX had the basics for a good group programming environment <5>.

By 1978, many groups had built coordination tools to support larger groups of people, SCCS was widely used, and *uucp* at least existed to provide some minimal communication facilities <5-10>.

By 1982, a number of groups had built much more extensive programming environment systems (such as SOLID or MESA inside Bell Labs), and mailers had become more sophisticated <5-15>.

In 1986, more powerful distributed system facilities are becoming available <20>, including some good non-UNIX ones by Apollo and DEC, and a good UNIX-based one by Sun Microsystems.

By 1990, we can assume that the best UNIX-based group programming environments will be as good <30>, and maybe better, than the best available otherwise, mainly by catching up with the graphics that it does not yet support consistently.

4.3. Bigger Problems

Another way to measure leverage is to look at the maximum number of people that could work together using UNIX as a development base. This mostly derives from the size of machines and the internetworking thereof. The numbers below give approximate maximum sizes.

Leverage Summary

1974		1978		1982		1986		1990	
UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest	UNIX	Rest
25	200	100	400	400	500	500	500	1000	1000

In 1974, project size was pretty much limited to a group of people who could fit on one PDP 11/45.

In 1978, a project could survive the use of several close-linked 11/70s. Some large projects simply could not fit onto the available UNIX boxes.

By 1982, larger projects used linked groups of VAXen. Really large projects (like #5 ESS) were moving onto large mainframes, using UNIX/370.

In 1986, UNIX runs on the largest computers. Even the largest projects can use UNIX for their development support.

In 1990, I assume that truly immense projects can be UNIX-based if they so desire. I would hope that other areas of leverage have improved enough that we could do immense projects with fewer people, but I suspect that people will simply try to do even larger projects.

5. Reusability of Code

Not writing new code is the ultimate in programming leverage. Reusability has 2 separate aspects. First, there must be something to reuse. Second, you have to be able to find it.

This is so subjective that I've omitted the numerical ratings. However, I'd observe that the common wisdom of 1974 was to re-read the UNIX Programmer's Manual once a month. You always learned something new, and besides, it didn't take very long to read it.

By 1978, this was harder, and it's unthinkable today. Our ability to organise software and make it available has lagged behind our ability to write it, and we've yet to achieve Doug McIlroy's vision of software component catalogues ["Mass Produced Software Components", NATO Software Engineering Conference, 1970].

Another way to say this is to paraphrase Maslow's comment on tool-poor environments:

``To the person with only a hammer, all the world looks like a nail.''

Unfortunately, in a tool-rich environment that has not kept its tools superbly organised:

``For the person with a giant workshop, it may take longer to find the right screwdriver, than to take the hammer, rip the screw out, and be done with it.''

I hope to see substantial progress by 1990, using high-bandwidth interfaces and tools akin to Smalltalk browsers.

6. Portability

This may be the most important area of leverage, but is also the easiest one to describe. Quite simply, no other general-purpose operating system comes close to matching the portability of UNIX to different hardware architectures. Although UNIX has often sacrificed short-term performance for portability, it has often prospered in the longer run by being able to quickly move to new architectures. Consider how few other previously-existing operating systems have been able to shift to microprocessors.

In fact, in the next few years, we're likely to see an ironic payback to UNIX in return for its portability. Most older operating systems are tied to specific hardware architectures, whereas the existence of UNIX makes possible the creation of new architectures without requiring huge efforts to develop new operating systems. Thus, most new machines will run UNIX, and many may run nothing but UNIX. If it happens that big jumps in performance and cost/performance occur via new architectures, then UNIX will have a substantial advantage in gaining access to especially effective machines. Many people believe that RISC-based microprocessors are going to offer such big jumps over the next few years. (Since I work for a RISC microprocessor systems company, I might be accused of bias on this!) However, many companies are betting the success of major projects, or even their whole existence, on this possibility. Every one of these machines runs UNIX. This leads to one last prediction.

By 1990, the most cost-effective machines in the 5-50Mips range will all run UNIX or some closely-related variant.

7. Conclusion

In 1988, I think UNIX most needs to more widely incorporate the improvements in human interface that have appeared on personal computers. (This means that people who can spend $1000 on the desktop get graphics also, not just people who can spend $5000.) Next, we desperately need to regain the degree of reusability we once had, i.e., where you easily could find and re-use anything there was.

Finally, UNIX portability will offer an exceptional source for leverage over the next few years, as people will be able to move masses of software to much more cost/effective machines.

The result should once again show the wisdom of investing in long-term portability, rather than short-sighted optimisation.

Acknowledgements

This paper is an updated version of one that appeared in the USENIX Winter 1987 Proceedings.

Plan 9 from Bell Labs – The Network

David Leo Presotto

AT&T Bell Laboratories
Murray Hill, New jersey 07974
research!presotto
presotto@att.arpa

ABSTRACT

This paper describes a new computing environment and the networking that underlies it. We expect the environment to accommodate either a small group or a large organization. Although our initial implementation is targeted at 100 researchers, our goal is a system that can encompass all of AT&T's research and development.

Our design runs counter to the popular trend in computing environments, workstations connected by local area networks. We have found this solution to be both expensive and awkward. This is especially apparent in large organizations. Instead, we propose a system based on clusters of file servers and execute servers connected by high speed networks. User interfaces, similar to workstations, access the servers via lower speed distribution networks. Among other things, this simplifies administration and allows the home and work computing environment to be the same.

1. Introduction

About a year ago, Rob Pike and Ken Thompson started designing a replacement for UNIX. Several of us have since joined in. Our objective is to provide a system that can provide the computing resources for an organization the size of AT&T's R&D community. This translates into supporting approximately 20,000 users at dozens of sites across the United States. Currently this community is served by a number of different systems. Although they come from many manufacturers, they can be broken into two broad categories; large time sharing systems and workstations. It would be convenient if one of these categories could be used for our community wide environment. Unfortunately, there are problems with both solutions.

The large timesharing systems were built to centralize administration and maximize sharing of both equipment and information. When the cost of user terminals was cheap relative to the rest of the system, this was a very cost effective arrangement. Unfortunately, the need for graphic interfaces has driven the cost of the terminals up while vlsi has driven the cost of processors down. For little extra cost, processors can be added to the user terminals creating personal computers or workstations.† This has made it, in many cases, more cost effective to abandon the time sharing system in favor of the workstation. Since the market factors forcing the change continue, the percentage of systems to which this argument applies increases every year.

Users who have migrated to workstations face the problem of sharing their more expensive resources. Those most often shared are long term storage and hard copy devices, the former because information sharing is a goal in itself, the latter because of financial reasons. The obvious solution is to join the personal computers via cheap local area networks (LAN) providing a medium over which the storage and printers can be shared. These network systems are normally pieced together from off the shelf parts (UNIX[1], TCP/IP, ethernet, whatever micro is currently in fashion, a remote file system, ...). Using broadcast algorithms and cheap media (e.g. ethernet) these networks can be comfortably grown to 100 or more workstations.

The disadvantages of workstation networks are less obvious. Because of the small incremental cost when getting started, the workstation LAN seems rather inexpensive. However upgrade costs can become

† We consider personal computers and workstations to be essentially the same thing.

astronomical. Since the model tends to couple the processor with the graphic interface, each user is trapped in his own little box. Although the processor in most of our current boxes is more than adequate for the graphic interface, we have to constantly upgrade each box to deliver more computing power. Our time between upgrades for AT&T 3B's, Sun's, and DEC microvax's seems to be about 2 years. The upgrade cost of each system runs from $10,000 to $20,000. That means an expense of $500,000 to $1,000,000 a year for a 100 person laboratory.†

A second and more fundamental problem stems from the bandwidth needed by the traditional workstation environment. It has already become difficult to take our diskless workstations home with us. The bandwidth available over LAN's (typically 10 megabits) is unavailable to the home. This means that our home and work computing environments cannot be the same. The typical solution is to add disks to the home systems. Unfortunately, this increases cost and reduces the possibilities of sharing. The same argument also applies to the newer high speed workstations. Systems as fast as SUN 4's are already too fast to run diskless on ethernets without being disk starved. This too requires a faster network or private disks.

The only way around these problems is to provide a few execute servers that anyone can use. By upgrading these servers often, we can avoid upgrading a much larger number of workstations. Unfortunately, few of the available networked systems allow graceful access to execute servers. The environment in which a program executes on a remote server is almost always different than on the local system. This is true both in commercial systems like the various Unix's (Sun, AT&T, BSD, Ultrix) and even in research systems (Stanford's V-kernel[2], Xerox PARC's Pilot[3], Wisconsin's Arachne[4]). File names, variables (like the shell environment variables), or devices can be different. Only systems like Sprite[5], Locus[6], and the Newcastle Connection[7,8] which offer a true global name space seem to avoid the problem. Unfortunately, a large global name space can become rather difficult to navigate. A clear indication of this can be seen in the pseudo device mechanism in TOPS and the path variables used by UNIX shells. Both mechanisms are attempts to shape the name space seen by the user. Unfortunately, both are awkward in that they don't really change the name space but actually provide search paths available to some, but not all, applications.

The rest of this paper gives a birds eye view of an alternative being developed at Bell Labs. The project is in its infancy and most of what is described here is still under development.

Overview

Figure 1 depicts the topology of Plan 9. The Plan 9 system consists of a three tiered network with communal servers at one end and terminals at the other.

The servers are multiprocessor machines acting as either file or execute servers. The execute servers can be clustered together to replace what are now computer centers or can be owned by individual departments. Each cluster is connected by a local area network that provides 24 megabits/sec simultaneously between each execute server and file server in the cluster. The speed was chosen to approach that of the disks on the file servers in order to keep from starving the execute servers.

The terminal hardware is the equivalent of what most would call a diskless workstation. Our initial implementation is a 3 MIP machine with a 512x512 grey scale, screen, network interface, and mouse. By performing all the data intensive work on the execute servers, the speed of the terminal network can be kept low. Our terminal network is a 1 to 2 megabit/sec distribution network for our long haul network. The bandwidth to a single terminal is about .8 megabits/sec. Our experience with Blit terminals[9] indicates that this speed is adequate for host to terminal communications. At speeds as low as 1200 baud our Blits perform bit-mapped graphics equivalent to that of a diskless workstation. The limitation is the painfully long times need to transfer program and pictures over RS-232. A typical example of a large Blit program is our window based cut and paste editor, sam. Sam is 51000 bytes long and takes a little less than a minute to load over a 9600 baud line. A 512x512x2 grey scale picture is 64k bytes and takes about the same time to load. To make the response acceptable, we need a network that will bring these times down to a second or less. This means a bandwidth at least 60 times 9600 bits/sec or .6 megabits/sec, well within the bandwidth of our terminal network.

Finally, a wide area network connects the terminal and server networks. If we are to serve a nation-wide community, we need a network that spans the country. Our current long haul network provides 8

†Of course, we could (and do) just upgrade the 'important' people and let everyone else make do with the dregs. However, the 'less important' people don't seem very satisfied with this approach.

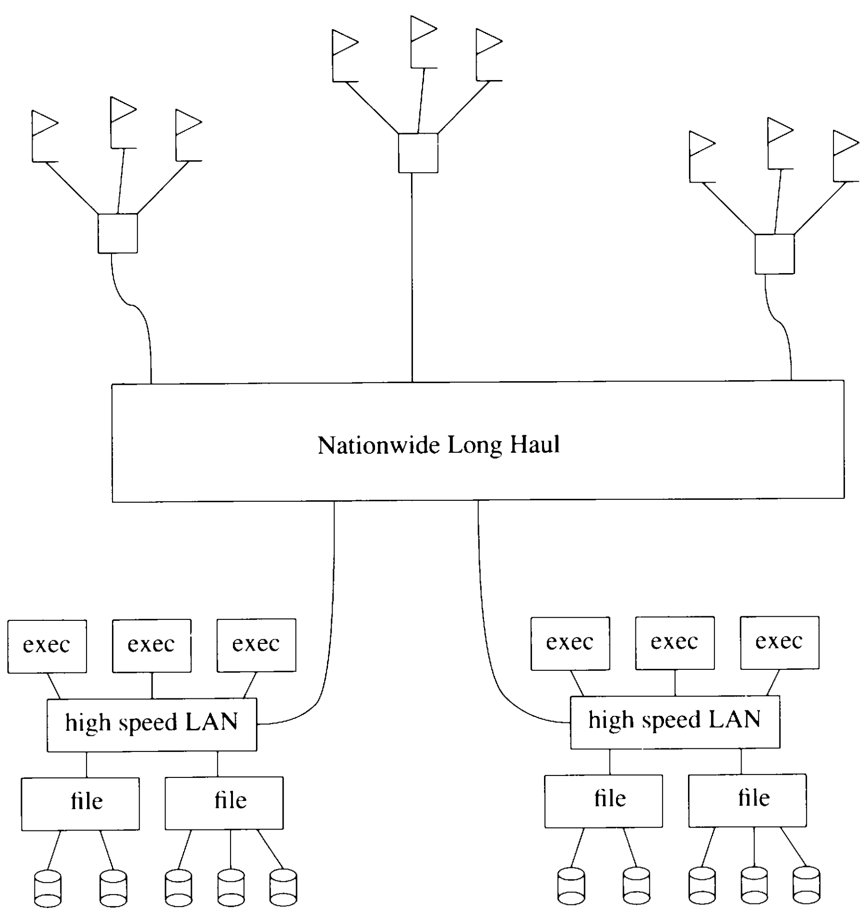


Figure 1. Plan 9 Topology

megabit/sec service locally (within a campus) and T1 trunking, 1.54 megabit/sec, for long distance. This should be a good first solution to our long distance networking. A higher speed, 125 megabit/sec, version is being planned with T3 trunking, 45 megabits/sec.

Plan 9 melds the advantages of time sharing systems and workstation networks without their disadvantages. Like a timesharing system, the file servers and execute servers can be shared by all users but are administered centrally by a computer center. Like a workstation environment, any user or group of users can attach their own execute and file servers to the network and reserve it for their own use. In addition, each user can choose to run processes either in their own terminals or in the execute servers. A versatile name space makes all this possible.

Name Space

In Plan 9 all named objects are in the same global name space. Processes, files, environment variables, network connections, and devices all look like files or directories of files. The only interface to these objects is via mount, unmount, open, close, read, write, and seek. This is a direct descendant of UNIX device files and the ninth edition UNIX file system switch.

The structure of the object name space is a forest. Any resource connected to the network (e.g. terminals, file servers, or print servers) implements a tree of the name space. The resources themselves are named in a separate network name space described below. Whenever a user logs into Plan 9, he first creates a personal name space on his terminal by attaching to his terminal's tree whole trees or subtrees from other network resources. The first subtrees he must attach are his home directory and standard utilities. This is performed by the login process when he authenticates himself with the network. Other parts can be automatically added (or removed) by a profile in his home directorv.

Once the user has a name space created, he accesses objects in that name space by reading and writing them as if they were files, as they may indeed be. Objects like the user's environment variables, screen, keyboard, mouse, and processes are implemented by the terminal itself. Objects like his home directory are actually being remotely accessed.

New processes inherit the name space of their parent. Any changes made to the name space are visible to other processes in its process group. If a process wishes, it can start a new process group. That process group starts with a clone of the old name space. Changes made to a process group's name space are not visible to other process groups on the same machine. Therefore it is possible for a number of users to be executing on the same machine with totally disjoint name spaces. This is especially important for execute serves. When a user creates a process on another resource (e.g., if a user opens a window to an execute server), a description of the parent process's name space is sent to the new processor. Before starting the child process an analog of the parent's name space is built for the new process. Since all accessible objects are in the name space, the child process will act exactly the same on the remote machine as it would have on the local one.

The reason we say that the child will have an analog of the parent's name space is because we may wish to perform some changes in the sequence of mounts defining the name space. For example, if the local system mounted /bin.68020 onto /bin the remote may want to mount /bin.cray2 onto /bin. This would be one way to provide for a heterogeneous processor environment.

Long Haul Network

Plan 9's networking is based on our own long haul network, Datakit[10]. Datakit provides the protocols and name space to be used on all three tiers. Datakit is a virtual circuit switched network of the same vintage as ethernet. The network consists of nodes (the switching elements), external devices (hosts and terminals), and trunks. All devices attach directly to nodes which are joined together by the trunks. The network provides virtual circuits between any two devices, a hierarchical name space for devices, and an authenticated source identifier for all calls.

Our choice of Datakit as a long haul network is essential to the design of Plan 9. Our remote file access protocol, like most other such protocols, is based on send/reply semantics. For Plan 9 to work across long distances, message latency must be kept down. At the moment only virtual circuit networks can guarantee low latencies at distance. This is due, in part, to the ability of a virtual circuit network to turn away offered load when the delay through the network becomes too high. For example, round trip delays across the United States on Datakit are consistently 55 to 65 milliseconds. The delay for the equivalent path on the DOD Arpanet varies from 250 milliseconds to 10 seconds. If packet networks are to begin to provide latency guarantees, they will need to develop much more stability.

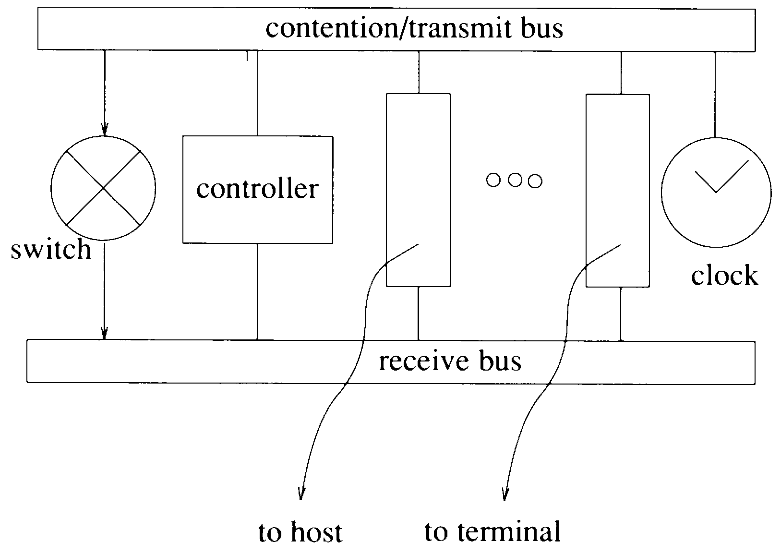


Figure 2. Datakit Node Architecture

Figure 2 depicts the architecture of a datakit node. Devices and trunks connect to the node via interface modules which plug into each of the two 8 megabit busses. All modules transmit on one bus and listen on the other. Packets are kept small (18 9-bit bytes) to reduce transmit latency and increase bus sharing. As

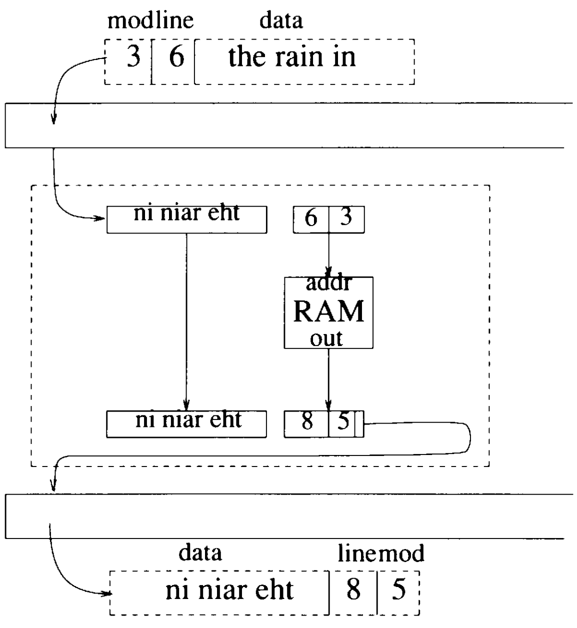


Figure 3. Datakit Switch

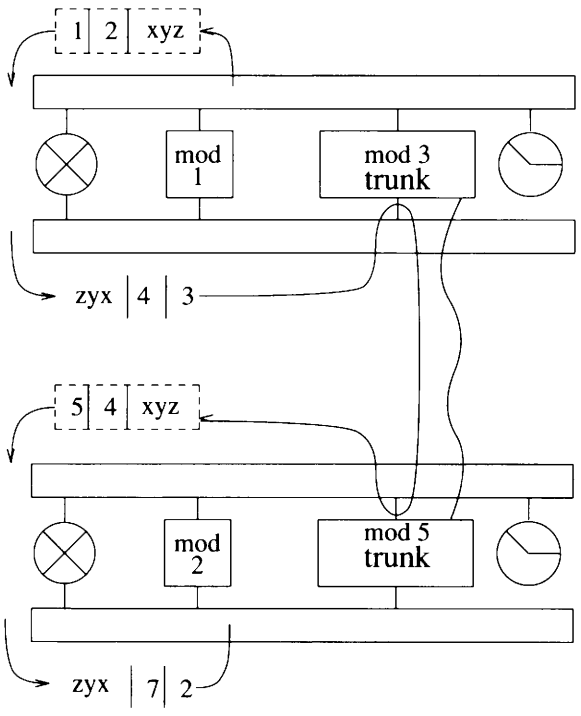


Figure 4. Trunking

figure 3 shows, each packet contains a source field containing a module number and channel number. As the packet passes through the switch on the way to the receive bus, the switch looks up the source field in a local ram and replaces it with a destination field. Trunking is accomplished by making the destination in the switch memory a trunk module. A packet that trunks across one hop is shown in figure 4.

In order to set up a call in switch memories for a virtual circuit, a terminal or host has to communicate with the node controller. For this, each interface has one virtual circuit permanently attached to the controller on its local node. When setting up a new circuit, the device making the call informs the controller of the channel to be used and the name of the destination. Then, communicating with other controllers, the local controller determines a path in the network. Finally, the switches are then set up in all relevant nodes and both caller and callee are informed that the call has been made.

The names space is a fixed height tree with all the devices at the leaves. Device names look like UNIX file names. They are variable length strings with components separated by slashes, e.g., att/nj/astro/research. Unlike file names there is no root and all names are relative. A name is resolved by walking up the tree from the current location as many elements as specified in the name and then using the name as a path through the tree from that point.

It is important to note that the physical topology of the network and the name space are completely disjoint, i.e., connectivity in the network is not related to the hierarchy of the name space. The former reflects the administrative structure of the network and the latter the flow of information in the network. That means that trunks can and are placed between any two nodes which need the bandwidth independent of their relation in the name space. However, in practice, the structure of the network usually coincides with that of the name space since administration and information flow often coincides.

Since all the switching is done in hardware, delays through the network are very low. Ignoring high load queuing delays, the time through a node is on the order of a millisecond. Also, since the path for a call is fixed, packets cannot pass each other. Together these properties make it easy to build very lightweight transport protocol. The transport protocol used in the network is the Universal Receiver Protocol (URP). URP provides grades of service ranging from uncorrected character streams to flow controlled corrected delimited streams.

Terminal Network

The terminal network is a distribution system for Datakit called INCON, for *intelligent connector*[11, 12]. INCON serves up to 15 stations over a balanced copper pair for 1000 feet at bit rates of 1.54 to 2 megabits/sec. Connection to incon is via connectors that convert the link level protocol of the connecting device to the link level protocol of INCON. For example, a standard ascii terminal connects via an RS-232 connector or a telephone via a digital phone connector. Hence the name, intelligent connector.

One or more INCON networks are connected to a Datakit via a *breaker box*. the breaker box can be either a module plugged directly into a datakit or a free standing module connected to a distant Datakit via a trunk. In essence the INCON network is a low cost, low speed extension of the Datakit busses. Therefore, the breaker box must map between INCON and Datakit addresses when transferring packets between the two. It does this using a map which is configured by the datakit controller and it is functionally equivalent to the Datakit switch with similar low packet delays.

In Plan 9, an INCON network will serve a single office utilizing the current telephone wiring. The INCON wires run to a telephone closet where they terminate in a breaker box which trunks over fiber to a Datakit. Home INCON's connect to a breaker box in the house with a 56kbs, 144kbs or 1.54Mbs trunk back to a company Datakit.

High Speed Network

Currently Datakit has no usable high speed component. A 125 megabit/sec version, called Hyperkit, exists in prototype form. However, we are still waiting for chips to be built to provide a high speed interface between it and our servers. The Hyperkit's architecture is the same as a Datakit's. It's speed is derived from making the busses wider and by performing transmit contention for cycle N+1 during cycle N of the bus.

Luckily, for the near future there will only be one cluster of servers containing only a handful of multiprocessors. As a stopgap, we are directly connecting all or our execute servers to all of our file servers. The connections use parallel DMA interfaces and provide 24 megabit/sec transfer speeds. In addition, each server has a datakit interface to connect to the terminals.

Status

The country wide Datakit network has been in place for some years. We have connected a dozen INCON networks to it, each carrying one of our Plan 9 terminals. The terminals are running an operating system with a VAX 11/750 is currently acting as their file server. A new multiprocessor is being installed as our first execute server.

We are starting experiments with home service to see what bandwidths will be necessary to the home. A number of us now connect to Plan 9 from home via dedicated 9600 baud lines. Rob Pike has just had a 1.54 megabit/sec trunk installed in his home and will soon be using that to trunk an INCON network to one of our company datakits. Soon we will install some 56k baud lines to get sample points in between. By the time this paper is presented, we should have some feeling for which speeds are good, which acceptable, and which useless.

It is too early for any conclusions. We are still feeling our way. If we've learned anything so far, it has been that networked UNIX is not the solution.

References

1. D. M. Ritchie and K. Thompson, ''The UNIX Time-Sharing System,'' *Comm. Assoc. Comp. Mach.*, vol. 17, no. 7, pp. 365-375, July 1974.
2. M. Theimer, K. Lantz, D. Cheriton, ''Preemptable Remote Execution Facilities for the V-System,'' *ACM Operating Systems Review*, vol. 19, no. 5, December, 1985. Tenth ACM Symp. on Operating Systems Principles
3. D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, et al, ''Pilot: An Operating System for a Personal Computer,'' *ACM Operating Systems Review*, no. 534790, December, 1979. Seventh ACM Symp. on Operating Systems Principles
4. M. H. Solomon, R. A. Finkel, ''The Arachne Distributed Operating System,'' *ACM Operating Systems Review*, no. 534790, December, 1979. Seventh ACM Symp. on Operating Systems Principles
5. M. Nelson, B. Welsh, J. Ousterhout, ''Caching in the Sprite network File System,'' *ACM Operating Systems Review*, vol. 21, no. 5, November, 1987. Eleventh ACM Symp. on Operating Systems Principles
6. B. Walker, G. Popek, R. English, C.Kline and G. Thiel, ''The LOCUS Distributed Operating System,'' *ACM Operating Systems Review*, vol. 17, no. 5, October, 1983. Ninth ACM Symp. on Operating Systems Principles
7. D. R. Brownbridge, L. F. Marshall, B. Randell, ''The Newcastle Connection - or UNIXs of the World Unite,'' *Software - Practice and Experience*, vol. 12, December, 1982.
8. L. F. Marshall, ''Remote File Systems Are Not Enough!,'' *EUUG Conference Proceedings*, September, 1986.
9. Rob Pike, ''The Blit: A Multiplexed Graphics Terminal,'' *Bell Labs Tech. J.*, vol. 63, no. 8, part 2, pp. 1607-1631, Oct. 1984.
10. G. L. Chesson and A. G. Fraser, ''Datakit Network Architecture,'' *IEEE Compcon '80*, January 1980.
11. R. C. Restrick, ''INCON Wire Interface Integrated Circuit Design,'' *AT&T Technical Memorandum.*
12. C. R. Kalmanek, ''INCON: Network Maintenance and Privacy,'' *AT&T Technical Memorandum.*

Help! I'm Losing My Files!

John Lions

University of New South Wales
Kensington 2033
Australia

ABSTRACT

Managing large collections of miscellaneous files can present a problem for individual users of a UNIX system. Keeping track of files that are still wanted and useful, finding and eliminating files that are no longer needed, and reorganising the file hierarchy from time to time may not be trivial if the set of files is large. Outlines are drawn for a partial solution involving index files, embedded keyword lists, a procedure for revising file pathnames and the implementation of a daemon secretary to keep everything tidy.

''Old programmers never die; they slowly file away''

1. Introduction

This paper is a personal call for help. I've been suffering from a familiar problem for years, and lately it's been getting worse. I'm losing files — not because of malicious hardware, or overzealous administration — I'm simply losing track of what are where, and it's getting harder to find them again.

Of course I'm not the first person to have this problem — for example, Dick Haight wrote *find* many years ago — but these days *find* takes 2-3 minutes of real time (35 seconds of cpu time) in my main account. Given that it takes so long, that I can't always remember enough of the name of the file I'm looking for, and that *find* doesn't help me browse, it is only a partial solution.

Experiences that have influenced this paper include:

1. I have been collecting files for a long, long time. In particular, I have been collecting files on UNIX systems for more than ten years. They are of diverse types (predominantly ASCII text) and currently reside under several user names on five different machines. There are many duplicates and near-duplicates (which is worse), and we are about to experience a major equipment upgrade.
2. For several years I have been pondering why the commercial success of UNIX systems has been less than many pundits have been predicting. Obviously the pundits have been naive: they simply got caught up in the enthusiasm of programmers for the UNIX system and extrapolated that to the environment of application users — not the same thing. But why? Part of the answer may be that the system supports the creation of many small files, many of which would be treated as single records in a standard database system. However it does not always provide a clear model for how to implement a workable, working application and perhaps there are still software tools waiting to be invented. It seems the UNIX system model may yet be incomplete(?).
3. Until recently, for nearly five years, in my tidier moments I was editor of the *Australian Computer Journal*. During this period I used my local UNIX system to maintain records of correspondence and work-in-progress. In the process I developed specialised solutions to what can now be perceived as just subproblems of a more general problem. Last August I gave a paper (Lions, 1987) describing these solutions to the AUUG meeting in Sydney. What emerged from the discussion was that the problem, particularly in its more general form, was more interesting than the particular solution techniques I had been using, so I decided to extend them.

2. The Problem

How does one manage a large assemblage of files that has been created somewhat haphazardly over a long period of time. (I am thinking particularly of my own personal collection. Suppose, for point of argument, one has 5000 files that can be ''browsed'' at the rate of 30 per hour. How long will it take?) Each of these files was valuable for at least part of its existence, and may be so still. Some may still have archival or just sentimental value. I still want to keep many of them, and to be able to find them easily. How does one keep the collection "properly ordered"? How does one cull the collection efficiently and effectively?

This problem will not be news to anyone who owns a filing cabinet. One tends to store things indefinitely as long as the marginal cost of keeping them is less than the cost of disposal. The cost of keeping computer files is now very low, and is still decreasing rapidly (so it is most probably already less than for paper files). For me the cost of culling online files, measured in terms of my time, is already greater than for paper files. Hence I can already assume that my file collection is likely to keep growing for some time yet. (A rate of 1000 new files per year will almost certainly be conservative.)

This will be acceptable as long as the clutter does not obscure those files that I really want. I have colleagues who strongly recommend the "*rm -r **" approach for all such questions but that's not my kind of solution. (Besides, it may not be appropriate if, for example, files are stored on non-erasable storage.) I want to be able to:

1. locate files efficiently using keywords or phrases that may or may not be embedded in the file name;
2. devise ways to rearrange the hierarchy for all or part of my files with little or no direct intervention from me;
3. find efficient ways to locate files that are no longer needed and should be deleted.

There are well-known database management system solutions for some of these problems, but they seem often overly rigid and constraining. Even if they support collections of text files, they usually want to impose structures and constraints on the files' internal structures. There already exist UNIX tools for rearranging the file hierarchy (especially *mv*, *cp*, *rm*, etc.) but they are time-consuming to use, potentially error-prone and too primitive for the problem now confronting me. I'm looking for a better way that will be consistent with basic UNIX philosophy: perhaps not perfectly general, but neat, tidy and effective. I'm happy to describe my solution in the hope that someone else has an even better idea.

3. A Partial Solution

The main problem associated with my editorial role was to navigate efficiently around a reasonable sized (c. 1600) collection of files. A typical morning's work might involve locating and interactively editing twenty or thirty scattered files to record events and changes, and to append commands for generating letters to be produced later by a background batch process. Such navigation problems arise frequently in office-style computer applications and conventional data base systems explicitly address these (for example see Bachman, 1973).

By offering good support for large assemblages of small files, the UNIX system provides a viable alternative to conventional data base systems for many purposes. If it is used properly, in many cases there is no need to use a proprietary database system.

I do believe that UNIX has a problem with supplying a wide enough range of role models. Although the standard system offers a strong, firm model for program developers, I submit that its implementation models do not offer broad enough guidance for application developers in specifying and organising large assemblies of small files.

I am more than happy with the solution that I have developed because it works well. This is not because it uses all the latest mouse/window technology (it doesn't) but because it works well *without* the latest rodents on a heavily loaded VAX system. In broad terms the method I have used is:

1. Establish a set of *domains* within the file hierarchy. Each domain corresponds to a broad category of files. (I formerly used the word "area" for the same idea, but "domain" now seems better.)
2. Associate each domain with a separate directory in the directory hierarchy. This directory becomes the domain's *home directory*. It can serve as the repository for all sorts of administrative files (including specialised commands) associated with the domain. It may also contain subdirectories.
3. As far as possible, locate the files that will be associated with the domain "close" in the directory hierarchy to the domain's home directory.
4. As far as possible, ensure that the set of domains completely covers the file hierarchy, i.e. each file belongs to at least one domain.
5. Establish a *name server* for each domain.

Various exotic possibilities can be conjured up for the name server, but as I currently see it (and certainly as I have implemented it) the service is provided via procedures applied to an *index file* that has one record for each file in the domain (including itself) and, for each such file, relates the file's pathname (relative to the domain's home directory) to a string of keywords and phrases that ''characterise'' the file in some way. For example:

> *relative_path_name:* *list of keywords*
> index:
> current/bills1: gas water electricity utilities
> current/bills2: dr smith boots medical

Finding a file involves searching the index file (using *grep* or *sed* is fine in my application) to find a record that matches a search string. The search string can match any part of a record. Matching the file name (the part before the '':'') is usually better because it is more likely to result in a single match. But using a more general term may often be the only available alternative. If more than one index record is matched, then a further selection can be made from the information returned to the user.

Creating the index files involves deciding which files belong to the particular domain, and obtaining a string of keywords to characterise each individual file. One way to do this is discussed below.

Once the initial version of name service was working, finding files for editing became much easier. For example, if I needed to change the file for an article whose title included the word ''concurrent'', then, knowing that ''article'' is the name of the relevant domain, and that the keywords are taken from the title and the author's name, I could start by executing the command ''**edit article concurrent**''. My version of *edit* is a shell script that:

1. changes directory to the home directory of the domain named by the first argument;
2. searches the index file for the domain for occurrences of the string named by the second argument;
3. if the standard output is to a terminal, displays all the matching records. If there is more than one, the user is then asked to supply a further search string that can be used for making a second choice;
4. extracts the file name from the beginning of the index record, expands the file if it has been compressed, and starts an editor (*vi*) for it.

Edit provides a convenient way for moving around the directory hierarchy and explicit use of *cd* has been greatly reduced as a consequence.

Similar commands were devised for other purposes. The command *fs* simply searches the index file denoted by its first argument and returns the records matching the string that is its second argument. Both commands are useful. Although it would be possible to use e.g. ''**vi `fs article concurrent`**'' in place of ''**edit article concurrent**'' the lack of the implied current directory change makes the former much less attractive.

Edit offers an attractive way to find and access all the other files as well. In particular, interrupting one editing task to perform another e.g. to revise the text of a standard acknowledgement letter, is frequently desirable, and so executing commands like ''**edit letters acknowl**'' becomes desirable. But such commands wouldn't work unless a domain for ''letters'' has been established with its own index file. To do this, a new way of creating index file records is needed, since the previous way, using a *sed* script to munch on standard record types in the data file, no longer applies.

4. Index File Records

Creating an index file was quite straightforward initially. The original domains coincided with selected major subtrees of the file hierarchy; a file was in the domain if and only if it was part of the subtree. The relevant files in each domain had a well-defined internal structure (in fact might have been a database record in a more conventional system), and was given its own index file record. These records could be, and were, derived mechanically using *sed* to search for and combine titles and author names.

When the concept of a domain was widened, purely mechanical means for generating index records became impractical (if not entirely impossible). My solution was to introduce a new type of record into each standard letter file, and the other files as well. One or more records of this new type need to be included in each relevant file. This can still be done mechanically for the standard file types, or by hand otherwise. The record type needs to be distinctive, so that it can still be *found* and trimmed easily using *sed*. The convention for the record type that I have established is:

- It consists an arbitrary initial string ending in the special, unlikely combination ''**#@$:**'' followed by the list of keywords to be associated with the file.

- It can appear embedded as a comment in all sorts of files, even an executable binary program (e.g. by including "**char** blah [ ] = "\n#@$: keywords\n";" in the source file).
- It can be generated by hand, or mechanically for data files in a prescribed format. In the latter case even though it does not strictly need to be embedded in the file, it is a good idea that it should be, so that it can be "hand tuned" later if desired.
- It lives near the end of the file, front or back, so it can be found easily. (If this can be enforced, then a special variant of *grep/sed* could be usefully devised for extracting such records quickly.)

The string "#@$:", which was chosen for its expected rarity, is unpronounceable (except in comic strips) so the new type has been called a *blah* record. It carries information that many would regard as a *file attribute*, i.e. "out-of-band" information that should be represented outside the file itself. (One can safely assert that to do this would be inconsistent with general UNIX philosophy and hence should not be considered a possibility.)

Blah records usually have to appear as comments in their host file. This is no problem for types of files where "#" is already used to introduce comments e.g. shell scripts. Alternatively, the record may be prefixed by ".\"" if the file is *troff*able, or surrounded by "/*" and "*/" if it appears in a C program.

The need for variant types of blah records depending on the file category is an irritant when records are being installed manually. Conventions for comment record types in most situations are firmly established and cannot be revoked. However it would be useful if a "super comment type" could be established to be recognised in addition to the already established comment types for a wide variety of file categories. (I put this forward as a suggestion for consideration by standardisation committees.) Since the C preprocessor already recognises records beginning with "#", it would seem to be relatively simple to introduce "hash-led" comments into C programs, for instance.

I have embarked on a campaign to include blah records in many of my existing files. Since the collection of files is far from systematic, the blah records have to be hand-crafted. To aid this process, I defined a *blah* program with several options which can generate the substring "#@$:" reliably on request. I have also modified a "browsing" program to invoke *blah* to add the extra lines at the end of the file and then to restore the previous modification date (very pertinent "out-of-band" information).

Blah records need to be derived using a restricted, standardised vocabulary or thesaurus. I do not underestimate the difficulty of doing this, since there are well-known problems in developing and maintaining list of indexing words (for example see Judge and Gerrie, 1986, p.107 & p.149*ff*).

If we can suppose that the problem of associating list of keywords with files has been solved satisfactorily (e.g. using blah records), then the problem of creating index files for domains can also be solved, provided the association between files and domains has also been defined. This association may be defined by the relative positions of files and domain directories in the file hierarchy. Or it may be defined in some other way.

5. File Pathnames

An idea that has more than just a passing attraction is to derive some or all of a new pathname for a file from its keyword list. Such a derivation may be complex, and subject to idiosyncratic rules in particular cases. It may also need to change as the directory structure changes, as old directories are removed and new ones added. This suggests new possibilities for *automatic filing* i.e.:

– moving a new (or newly imported file) from a general location to a more appropriate location; or

– rearranging files in an existing subtree among the directories in a new subtree.

There is an emphasis here on *moving* files i.e. on assigning appropriate pathnames. It could be suggested that once a full set of index files is implemented real file names will be of little significance, and can be assigned arbitrarily. There are several counters to this: most files *need* only one name at a time, so why devise an alias and then support a mapping between the two? When the time for "browsing files" arrives, the most natural collection to choose is usually the set of files in a particular directory.

The rules for the mapping "{**keyword list**} → {**pathname**}" for a file are potentially intricate and complex. At present I am experimenting with a simple program (called *rename*) to generate an alternative name for a file. It uses the following rules to generate a new name P for an existing file F:

1. Obtain the set of keywords for F and preserve their order.
2. From the beginning of the list, look for the first keyword K that corresponds to a domain name. If successful, set P to be the pathname for the domain corresponding to K; otherwise set P to the file's current domain pathname.

3. From the beginning of the list, let K be the first keyword that names a subdirectory of P. Replace P by P/K.
4. Repeat the previous step until no further changes to P are possible.
5. Replace P by P/B where B is the basename of F.
6. If F and P are the same, then exit.
7. While a file already exists with the name P, replace P by M(P) where M transforms a pathname in a suitable manner (e.g. by concatenating or modifying a numerical suffix).

This is sufficient to deal with imported files whose place in my file hierarchy has not yet been properly established, and to move files from an overfull directory down into a set of newly established subdirectories.

It also allows the reorganisation of subtrees when the attribute hierarchy is changed. For example, suppose a domain exists for *recipes* with two levels of subdirectories. Initially the primary classifier is *cooking style* so there are three level-two directories (*baked*, *fried* and *microwave*), and each of these is classified by *flesh type*, i.e. it has four level-three directories (*beef*, *fish*, *lamb* and *pork*). To reorganise the classification so that *flesh type* becomes the major classifier, the *recipes* directory is renamed e.g. *skip* and loses its index file. Then a new *recipes* directory is established with an incipient index file and four level-two directories (*beef*, *fish*, *lamb* and *pork*), each with three level-three directories, (*baked*, *fried* and *microwave*). The reclassification is then achieved by renaming all the data files below *skip* to move them back into the (new) *recipes* domain.

The mapping generated by *rename* is potentially multi-valued and and rules are needed for selecting among alternatives. It is also conditioned by the existing domain/directory structure and, in general, it will change if this structure changes.

Initial rewriting of the keyword list is needed in general. The components of the pathname F can be placed at the beginning of the list (this will ensure that once a file enters a domain, it will remain in there until the domain is abandoned). Other rules may be used to replace particular keywords or combinations of keywords by preferred alternatives, etc. In time, the keyword list transformer may come to resemble a major rule-based expert system, and be replaced by one.

6. The Secretary

Rearranging sets of files is a relatively slow and error-prone process. It is best performed at a time when other activities have been suspended. For my application where rapid action is not usually required, the obvious time is ''overnight''. I plan to implement a daemon *secretary* that will run at prearranged times. For each file that is designated to be moved, it will calculate the new file name and perform the move. Also, it will look for recently changed and/or moved files and make the appropriate changes to the index files.

A good secretary needs to know both what tasks should be done, and what should *not* be done. In this context, this means new conventions for directory hierarchies: that files in some directories should never be moved, and that any files in some other directories are essentially ''in transit'' and should be moved as soon as possible. (In deference to Dijkstra, I call the latter class of directories ''skip'').

7. Living With Imported Files

Even if indecision with how to classify particular files is a personal matter only, not to be replicated elsewhere, the problem of classifying files (records, data, etc.) is universal. Many of the files that concern me as I write this paper are imported, transported into my environment by *mail* or *news*. No one I know can possibly keep up with the vast flood of information that comes in over the news network, but only 90% of it is pure garbage. I would like a better way to cope, to select, store and manage more news items from the remaining 10% more efficiently.

Once I have proper keyword lists for imported files, I know I can do a better job of filing them. Mail handlers and news handlers should be adapted to facilitate the inclusion and editing of keyword lists (represented by blah records or whatever) for both incoming and outgoing items, and users should be encouraged to use them. An adequate keyword list should be required as part of every news item. Unfortunately Furnas *et al.* (1987) have recently confirmed the earlier experience of librarians that keyword selections by individuals may vary very widely when no standard thesaurus is enforced.

8. Other Relationships

Many of the matters discussed here have clear analogues in the problems that arise with addressing users across a large complex mail network. I borrowed the terms "domain" and "name server" deliberately, because I believe many of the present distinctions may largely disappear in time.

There is obviously a close relationship between index files and existing directory files. No suggestion is made here that their functions be merged, but perhaps index files might achieve some special status in future releases of UNIX systems.

Progress usually involves many small steps, not a few large ones. The solution described is designed to help me manage my personal set of files better without going to a full-blown database management system. It does not address issues such as concurrency, locking and recovery. I quite agree with Birrell, Jones and Wobber (1987) that the implementation has not been difficult.

9. Conclusions

This paper has canvassed the problems with managing a *large* set of personal files, a comparatively new problem for most computer users. A possible solution consistent with general UNIX philosophy has been outlined. This involves: the establishment and use of index files and domains to aid in locating particular files using keywords rather than precise file names; embedding a generalised class of comment records in files to record keyword lists; and implementing a daemon process that, like a good secretary, will keep everything tidy by moving files into appropriate clusters. Once I have finished implementing this solution for myself, I expect to stop losing files and to start finding them again!

10. Acknowledgements

Peter Chubb, John Hiller and Piers Lauder assisted in reviewing earlier drafts of the paper.

11. References

[1] Bachman, C.W. (1973): The Programmer as Navigator, *Communications of the ACM*, **16**, 11, pp. 653-658, November.

[2] Birrell, A.D., Jones, M.B., and Wobber, E.P. (1987): A Simple and Efficient Implementation for Small Databases, *Operating Systems Review*, **21**, 5, pp.149-154, November.

[3] Furnas, G.W., Landauer, T.K., Gomez, L.M., and Dumais, S.T. (1987): The Vocabulary Problem in Human-System Communication, *Communications of the ACM*, **30**, 11, pp. 964-971, November.

[4] Judge, P., and Gerrie, B. (ed.) (1986): *Small Scale Bibliographic Databases*, Academic Press, Sydney.

[5] Lions, J. (1987): What's in a Name? or Coping with Large Numbers of Small Files, *Australian UNIX systems User Group Newsletter*, **8**, 5, pp.78-83, December.

A Tool-based 3-D Modeling and Animation Workstation

Samuel J. Leffler

Eben F. Ostby

William T. Reeves

Animation Research and Development Group
Pixar
3240 Kerner Blvd.
San Rafael, CA. 94901

ABSTRACT

A tool-based system for 3-D modeling and animation is presented. Each *tool* is a separate program that operates as an independent UNIX process. Tools utilize a window-oriented display package, an event-based input system, and a large graphics database that resides in shared memory in providing interactive and non-interactive functions. The system described here is being developed for use in the production of 3-D animated sequences and as a testbed for research in 3-D modeling and animation. The architecture of the system and the motivation behind the tool-based approach is described.

1. Introduction

The Animation Research and Development group at Pixar is interested in the problem of creating realistic, computer-generated, moving imagery. In particular, we are interested in using computers to model, animate, and render life-like 3-dimensional characters in a feature length film. Many of the problems that must be solved to reach this goal are monumental. Aside from the computational difficulties associated with generating the pictures for a feature length film, the techniques required for modeling and animating life-like 3-D characters and scenery are either non-existent or extremely primitive. Present approaches to modeling and animation are typically very tedious, requiring so much human intervention as to make even short films extremely time consuming. Advanced techniques for improving the animation and modeling processes, such as dynamics and constraints are, at present, too computationally expensive to be considered for use in an interactive setting.

This paper describes an ongoing research effort to provide a framework within which production-quality modeling and animation systems can be constructed. These systems are constructed by combining individual programs termed *tools* that provide limited, but well-defined, facilities. This tool oriented approach provides users with a flexible and extensible system in which to model and animate. While one goal of this effort is to provide a production-quality environment in which to model and animate, a secondary goal is to create a testbed for experimenting with advanced techniques for modeling and animation. Techniques such as constraints and physical simulation are understood to be important in modeling and animation and it is important that it be possible to integrate them into our modeling and animation facilities.

The remainder of this paper describes the modeling and animation workstation being developed at Pixar. Section 2 presents, in reasonably simple terms, the typical process by which we create computer-generated 3-D imagery. In section 3 we discuss the goals of our system. Section 4 provides an overview of the system, including the hardware base on which the modeling and animation software operates. Section 5 describes the important components of the software structure that make up the foundational software layer we call the *modeling environment*. Section 6 discusses some of the tools that have been developed on top of the prototype system and, finally, in section 7 we summarize the work and describe future directions.

2. The Modeling-Animation-Rendering Process

The creation of realistic, computer-generated, moving, imagery is a difficult and time consuming process. Logically this process is divided into three steps: *modeling*, *animation*, and *rendering*. (We ignore the non-trivial tasks of designing a scene and the actions of the characters in the scene, and of creating the final film.) The first step in the process, modeling, creates a database describing the objects that are cast in a scene. These objects are usually constructed from simple geometric primitives such as cones, cylinders, and spheres; as well as from less familiar primitives such as patches and splines. Modeling is carried out either with interactive systems or through a programming language interface. In either case, the result is the same: a database of geometric primitives is generated describing objects to be placed in the final imagery.

The second step, animation, defines the motion of the objects modeled in the first step. Various methods are used to bind a database of time-varying values (the *motion database* or *cue sheet*) to the model. In some systems, each transformation node in a model is associated with from one to six channels of time-varying data; the binding is then implicit in the structure of the model. In other systems, the animator or modeler has control over which aspects of the model may be animated. In our system, time-varying variables, termed *articulated variables*, are incorporated in the model description. These variables may be used in expressions that are passed as arguments to geometric primitives or transformations. Each articulated variable corresponds to one channel in a database of time-varying values; with each channel independently specified by the animator. To minimize this specification effort, our system describes these channels as splines. Control points for the splines are termed *keyframes*, and the spline-fitting procedure used to determine control values at intermediate points in time is termed *in-betweening*. The in-betweening of keyframes is analogous to the techniques used in conventional cel animation.

Once modeling and animation have been completed, the user has a 3-D scene from which 2-D pictures can be generated. A *camera specification* is formulated to describe the position and field of vision of a simulated camera, as well camera-related parameters such the focal length of the camera's lens. Given a camera specification, images are then rendered at every 1/24th (for film), or 1/30th of a second (for video) during the time a scene plays. Scene information such as light sources and texturing are also provided to the rendering process.

As one might expect, the process just described is *iterative* in nature. For example, it is common for animation to show up weaknesses in a model, requiring alterations to the model that, in turn, require re-animation of portions of a scene. Typical 3-D graphics systems split the above process into three entirely separate steps. Furthermore, the programs used in each phase commonly utilize different, sometimes incompatible, databases. This can cause significant time delays when one must violate the *pipelined* nature of the modeling-animation-rendering process.

Of the three tasks described above, the process of rendering is by far the most advanced. Techniques for the realistic rendering of very complex scenes constructed from geometric primitives have been formulated. While the rendering of phenomenon that are not easily modeled with geometric primitives is less well understood, it is still advanced enough to permit the generation of very convincing images. Furthermore, with rapid advances in hardware technology, the time required to render images is quickly shrinking. Unfortunately, while the cost of rendering is dropping precipitously, the costs for modeling and animation have changed little. As a result, we are faced with the possibility that very soon almost all the time devoted to the creation of computer-generated imagery will be spent modeling and animating.

3. Design Goals

Interactive 3-D modeling and animation systems are computationally intensive. The manipulation and display of 3-D geometric primitives can, by itself, be very expensive. Combining this with database-oriented tasks and the handling of real-time input devices such as tablets, often swamps even powerful computers. While it has always been attractive to partition modeling and animation systems into multiple processes, the size of the geometry databases and the requirements for sharing this information have rarely made this feasible. Consequently, previous modeling and animation systems have usually been constructed as independent monolithic programs that are used sequentially and which communicate through data files that are defined with compatibility, rather than efficiency, in mind. Because the need for a common data format is at odds with the need for efficient access to the geometry database, programs are forced to convert input data to an internal format that is most efficient for their needs. This conversion can be very expensive, sometimes requiring hours for sizable ASCII databases.

Our previous systems for modeling and animation,[1, 2], exemplified these problems. Models were typically specified in ASCII using an algorithmic modeling language. Models and cue sheets were then interactively manipulated by the animation system to generate the motion database for the rendering system. If a model needed to be changed, users were required to exit the animation system, edit the model specification, and then restart the animation program. Similarly, many alterations to the cue sheet were not possible from within the animation program. Instead an ASCII representation of the cue sheet had to be edited off-line. While the text-oriented nature of these data files permitted the use of normal UNIXtext processing tools, their format resulted in lengthy delays in the startup of individual programs when sizable models and/or cue sheets were involved.

In designing our new system we wanted to address all of these issues as well as insure that the systems were extensible both for users and for developers. The overwhelming complexity of the previous monolithic programs led us to consider a tool-based approach similar to the scheme we had successfully used for raster framebuffers. In this scheme a central *modeling and animation framebuffer* is kept in stable storage. This database is manipulated with simple tools that can be combined in arbitrary ways to perform complex tasks. While tools in our raster toolkits have typically been used in a serial fashion, we envisioned the concurrent use of modeling and animation tools due to the interactive nature of the work.

The following sections discuss some of the important goals in our design.

System Extensibility

The system must be easily extensible both in terms of software and hardware. Since tools operate as independent processes, new facilities can be added through new tools rather than by altering existing tools. Even when existing tools must be altered, their simplicity and independence makes them easier to modify.

Hardware extensibility refers to the ability to support various input and output devices. In our environment it is common to have 2-D digitizing tablets of varying sizes, 3-D digitizers, knobs, mice, and so on. For performance reasons, our previous systems typically read directly from the input devices and were aware of the characteristics of the underlying hardware. We wanted tools to be shielded from the physical characteristics of input devices.

With regard to output devices, we were cognizant of the changes occurring in display technology. While raster polygon based display systems are still not fast enough to satisfy our performance requirements, they are improving at a rate that will soon make them usable. To shield programs from the details of the display device our graphics library is designed to be device independent.

Operational Flexibility

The system must be flexible to use. By partitioning facilities into independent tools a user may instantiate or combine tools in any way that suits them. For example, one of the most important tools is the *camera tool*, a tool used to display the contents of a scene on an output device. In most integrated systems, the camera control facilities are not under complete control of the user. That is, a user can not obtain multiple camera views of a scene unless the application expressly supports this. With our approach however, multiple views are created simply by running multiple instances of a camera tool.

A second consequence of using tools is that users can configure their operating environment as they see fit. Unwanted or unneeded tools can be iconified or removed from the screen. With only a limited amount of space on a display this turns out to be a substantial benefit.

User Interface Consistency

The user interface must be consistent across tools. This is a well understood notion that nonetheless requires careful thought. Simply specifying common menu packages is not sufficient. Interaction techniques in a 3-D workspace are not easily generalized to be applicable to all tools. We set forth certain ground rules for user interaction. For example, rather than have tools interactively support modal behavior, we expected users to simply startup another tool that provided the desired operating mode. Any modal behavior, of course, is expected to be accompanied by reasonable user feedback.

High Scene Complexity

The system must be capable of supporting scenes of extreme complexity. Complexity in our environment can appear in many forms. Scenes can be composed of many objects. Objects can be comprised of many graphical primitives. The display of a scene can require many lines or polygons. Models can be constructed with many graphical transformations. Animation can require many controls.

Our intent was to decouple display issues from those associated with the underlying models. That is, our design allows us to support models that are significantly more complex than we can comfortably manipulate in an interactive fashion. This allows us to be one step ahead of the display technology. In the meantime we can view overly complex models and scenes in two ways: either by reducing the detail with which objects are presented, or by displaying only part of a scene.

Procedural and Non-procedural Interfaces

The system must support both procedural and non-procedural interfaces. We believe, based on previous experience with modeling languages, that pure database-oriented approaches to modeling are severely limited. The ability to specify a model procedurally permits a concise and exact specification technique that is usually very difficult to achieve with non-procedural interfaces.

Central to our system is a graphical modeling language. Models are described with either textual descriptions (programs), or via interactive systems that generate model language descriptions. Internal to the modeling environment the modeling language is maintained in a more database-oriented structure. This internal form permits us to efficiently interface non-procedural system such as constraint solvers and simulation programs. There is, of course, the possibility of losing significant information, such as comments, when converting between an external procedural description and an internal database-oriented representation.

Interactive Performance

The system must be provide reasonable response for interactive tasks. This is a limiting factor in almost everything done in our system. As mentioned previously, we wanted to include physical simulation systems and constraint solvers in the interactive portions of the system; however, due to the limited compute power available it was not expected that this would be possible. Nonetheless, we have tried to insure facilities of these sorts can eventually be integrated with the interactive parts of the system. In the meantime, we plan to integrate non-interactive systems by using them to generate data that is manipulated interactively. For example, a physical simulation system might be used to generate motion descriptions which are then used as the basis for animation.

4. System Overview

The modeling and animation facilities provided to a user are built up from multiple applications that operate concurrently. Each application, termed a *tool*, provides a limited but well understood service to a user. Tools may be instantiated multiple times and combined in an arbitrary fashion.

Each tool is provided access to the workstation's displays and input devices, as well as access to a large shared database termed the *workspace*. The workspace is intended to hold all the information related to an animated scene; this includes:

- scene composition (models, cue sheets, cameras),
- model representation,
- symbol definitions,
- animation controls,
- display state,
- camera controls, and
- rendering controls (such as lighting styles and locations).

Since the workspace is potentially very large (many megabytes) and applications need immediate access to large portions of the data, a shared memory region is used to hold the workspace. A workspace is a long-lived object; it is possible to save the current contents of the workspace in a file and restore it at a later time.

Dividing information and state along the lines of a scene turns out to be worthwhile and has been developed as a general abstraction. Tools are designed to save and restore their operating state on a scene by scene basis. This facility, combined, with saving and restoring the workspace from a file permits users to easily checkpoint and resume work on a scene by scene basis.

The modeling and animation tools are constructed on top of a foundational layer of software termed the *modeling environment*. This software includes:

- a graphics package for drawing on calligraphic display devices,
- a package that provides event-oriented input facilities,

- several packages for the support of user interfaces,
- a model specification system,
- implementations of graphical primitives,
- a package for manipulating splines,
- a library for working with cue sheets,
- support for various styles of cameras, and
- a symbol table management package.

The graphics and input packages have been explicitly designed to support multiplexed access to resources. Thus, users are presented with an interface that looks much like a conventional window system; see Figure 4-1.

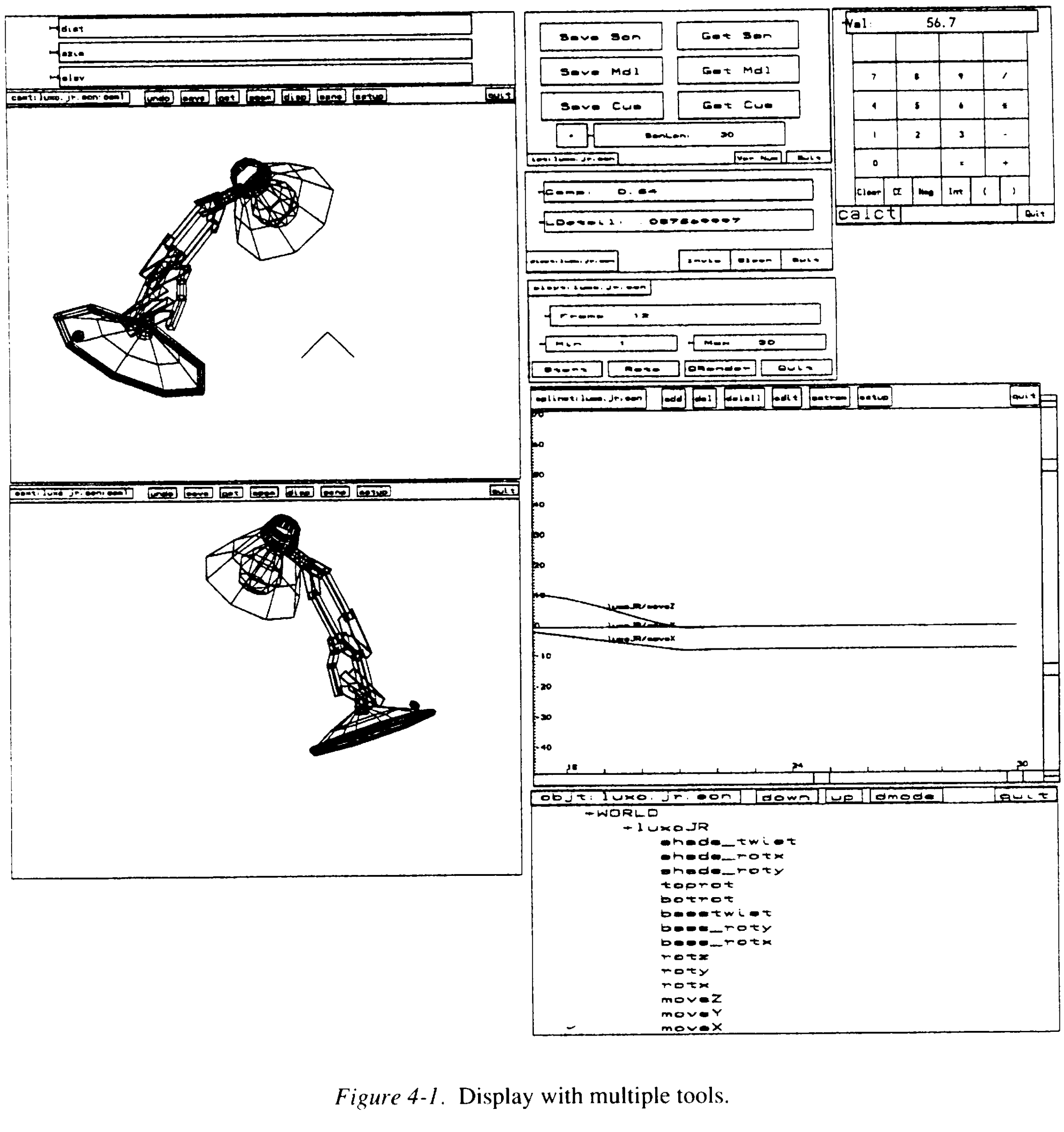


Figure 4-1. Display with multiple tools.

Hardware Base

Our prototype system is constructed from the following equipment (see Figure 4-2):

- 5-6 MIPS CPU with floating point accelerator,
- 16 megabytes of main memory,
- calligraphic display device capable of drawing 25,000 2-D vectors at 24 frames/second,
- monochromatic bitmap display,
- digitizing tablet for interaction.

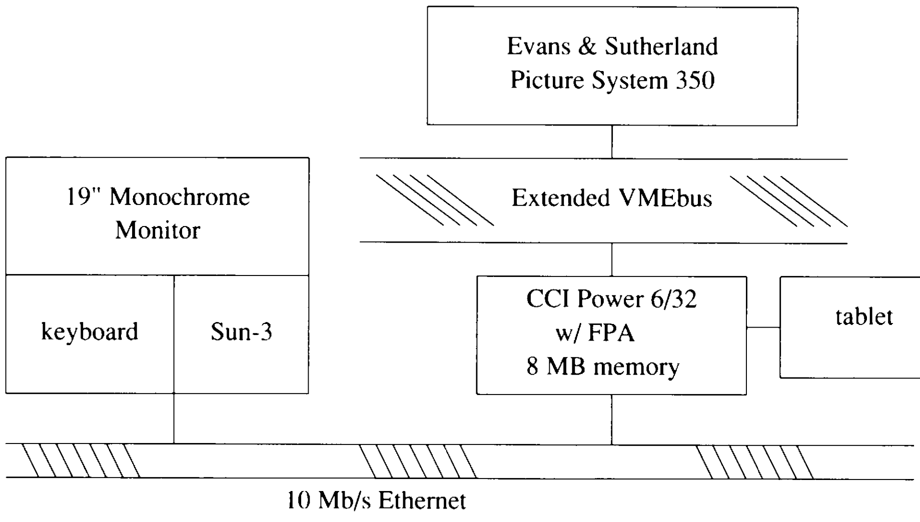


Figure 4-2. Workstation hardware configuration.

The hardware was selected almost three years ago. The CPU, reasonably powerful by todays standards, was selected because of the significant computational needs of modeling and animation. Our previous CPU, a VAX-11/750, severely limited the capabilities of our old systems. The replacement CPU was selected to be approximately 10 times as powerful (both in integer and floating point computation). We could, in fact, use unlimited amounts of CPU power. In animation, for example, the real-time solution of arrays of partial differential equations would permit animators to utilize physical simulation and rules of constraint.

The primary display device, an Evans and Sutherland PS350, is connected to the host via a special purpose hardware interface that we designed. This interface provides memory-mapped access to the internals of the PS350. In addition, the interface supports the capture and real-time playback of images,[3, 4]†. Our selection of the PS350 as a display device was based on its high performance (it is capable of displaying nearly 25,000 2-D vectors a second at 24 frames/second). We would have preferred to utilize one of the raster polygon systems, but were unwilling to settle for their relatively limited performance. We expect that, in the near future, the performance of these systems will improve to the point that we can ''step up'' from calligraphic displays. This will be a significant move as the presentation of complex graphical models is severely limited on line-oriented display devices; clipping and shading significantly reduces the ambiguity inherent in a vector display and increases the ''information content'' of an image.

The secondary display device, a Sun-3/50, provides a monochrome bitmap display that is used primarily for the presentation of auxiliary menus and for text. Sun's Network Extensible Window System (NeWS™),[5], is used for several reasons, not the least of which is that it was easy to ''drive it'' from a remote input device. In our situation the primary input device is a tablet located on the main CPU. The tablet is logically divided into multiple regions, each of which is assigned to either the main (PS350) display or to the alternate (Sun) display. When the stylus is in a region mapped to the Sun, tablet events are transmitted to the NeWS server and converted to NeWS events which are then dispatched as if they had

†This interface is designed to also work with the newer PS390 device.

occurred locally. This permits us to manage both screens through a single input device.

The Sun keyboard is similarly managed with a split-screen event distribution facility. Keyboard events are distributed according to the position of the primary input device. When input is focused on the PS350, keyboard events are passed from the Sun to the event distribution system on the CCI where they are then distributed.

5. Support Software

The system was developed on top of 4.3BSD. Two simple facilities were added to the kernel specifically for our needs: a shared memory facility that is used to hold the workspace, and a device driver for the PS350. Tools are written using a layer of software that forms the foundation for the whole modeling and animation system. This software was designed to support our needs in each of the following areas:

Shared memory. A single shared memory region is used by all tools. This region is located at a fixed address in each process's address space permitting tools to exchange pointers to data structures resident in the shared memory region. The support for shared memory includes memory allocation routines and synchronization primitives. One server process initializes shared memory and then remains active to insure the shared memory region is allocated (shared memory is reclaimed on last reference.) A second server process manages semaphores, reaping held semaphores from processes that terminate abnormally.

Graphics display support. The main display system is an Evans and Sutherland PS350 device. This display is supported by the CPAC2 graphics package, a device independent display list-oriented graphics package,[6]. CPAC2 is notable for its support of independent drawing *surfaces* termed *canvases*. Canvases provide an encapsulated drawing environment for each application and are organized in a hierarchy that is used in the distribution of input events. Canvases are allowed to overlap, but they are not clipped (on the PS350.) Finally, canvases may be *mapped* or *unmapped*. Mapped canvases are visible on the screen, while unmapped canvases are not. The latter is particularly useful for maintaining the display state of tools that are made iconic (with their icon displayed on the bitmapped display.)

Input handling. Input devices are managed by a server process that converts hardware events into software events that are dispatched to clients according to *interests* they have expressed. Processes may also generate software events to be distributed to other processes. Events are passed by reference through shared memory. The design of the event handling facilities was heavily influenced by the event handling facilities provided by NeWS[7].

User interface. There are several areas in which packages have been developed to provide a common user interface across applications. The common window management functions are provided by a window manager process. Each application identifies a region of a canvas in which the window manager should listen for input events. Using this mechanism, the window manager provides standard functions such as changing the shape and/or position of a canvas through a pop-up menu.

Interactive graphical *widgets* such as buttons and sliders are provided by an *item library*. Four basic items termed *buttons*, *toggles*, *sliders*, and *messages* are provided. These items can be combined to form familiar objects such as menus through the use of composition items termed *choices* and *panels*. The item library is widely used by applications resulting in a consistent interface.

The item library is commonly used in conjunction with the *layout library*. The layout library provides two facilities that are layered directly on top of items: an ASCII specification file and placement algorithms reminiscent of the *pic* and *tbl* programs. One useful aspect of the layout library, which is not yet fully exploited, is the ability to write out layout specification files according to an existing set of items. This permits applications to exactly save a large portion of their display state automatically.

Model language. The model language interface is available to all applications through the model language library. Tools have been written, using this library, to read and write models in an ASCII format. The model language is a fairly vanilla programming language oriented towards the specification of graphical objects. Basic control constructs and expression operators are provided as well as APL-like data structuring facilities. Graphical operations and primitives are not part of the model language per-se. Instead graphical primitives such as sphere or torus, are treated as opaque objects and treated in an object-oriented fashion by the model language library. A formal specification has been developed for the interface between graphics primitives and operators, and the model language. This interface definition permits primitives to be treated independently of the language.

The model language interpreter is one of the central components of the modeling and animation environment. Models are managed internally as interpreted data structures. Each time a model is *executed* the resultant calls to the graphics primitives that make up the model cause the model's representation on the display to be updated. Since the execution of large models can be extremely expensive, the model language library maintains dependency information that indicates which model language statements depend on which articulated variables. This information is used to optimize updates when articulated variables are altered.

Graphics primitives. As mentioned above, models are described in terms of graphics primitives, or *gprims*. Gprims are the foundation of the modeling facilities. Models can be described only in terms of the gprims supported by the system. Gprims may have multiple *representations*. Each representation reflects the characteristics of an output display or process. For example, a line representation is supported by all gprims and used for displaying gprims on the calligraphic display device. Other representations support the conversion of gprims to polygons, bicubic patches, and to micropolygons (an object used in the rendering system).

Splines. The spline library supports the manipulation of various types of spline curves used in the in-betweening process (e.g. Hermite, Catmull-Rom). Routines exist for constructing and editing splines, as well as for performing partial evaluations and updates. Timestamps are used to minimize the cost of updates when used in interactive systems. The spline library is also used to support a package of routines that implement 3-D splined space-curves.

Cue sheets. Cue sheets are used to specify the values for articulated variables. A typical cue sheet specifies a set of articulated variables by name. For each variable a spline type and set of knots is given along with the time span over which the values are to be applied to the model. Programming facilities are also supported by the cue sheet interface so that common cue sheets can be included, and so that data can be replicated and cycled. These facilities provide a crude mechanism for reusing existing motions – this is the start of a "motion library."

Cameras. A camera describes a view onto a scene. It is described by its position, orientation, field-of-view, and the aspect ratio of the view. Cameras are treated as special purpose graphics primitives in models. They are special because of their non-standard requirements for update notification and for their specialized interaction controls. By incorporating cameras in models it is possible to have cameras that present views that are affected by local transformations and also to have animated cameras (e.g. a camera that follows an object along its path of motion.)

Scenes. A scene is composed of a model and a cue sheet. The scene library is responsible for managing the state of the resources that comprise a scene. Updates to a scene must be synchronized so that, for example, multiple tools can not simultaneously modify the internal representation of a scene.

Matrices. The matrix library provides a complete package of routines for the manipulation of matrices, vectors, and quaternions. This library is sort of the graphics person's ~~l~~lm.

Symbol table. A central component of the modeling environment is a symbol table that resides in shared memory. The symbol table is used to record variables declared in models, articulated variables defined for animation, the names of objects in the workspace (e.g. the scene name, the names of models), and more. Symbols are defined within a tree of name scopes. Name scoping is necessary to handle name clashes. Names in the symbol table are heavily used for rendezvous between tools. For example, each camera registers its name in a special scope and programs that need to "attach" to cameras locate the camera by consulting the symbol table.

Selection. Information can be passed between tools through a selection service. This service manages a simple repository of data that is implemented on top of the symbol table package.

NeWS Clients. The final package of interest is a library of routines used by programs that communicate with the NeWS server managing the bitmap display. The library supports the basic NeWS facilities as well as some local additions that provide for sophisticated text manipulation. The library is described in[8].

6. Tools

As described previously, users interact with the modeling and animation facilities through tools. Tools typically cooperate with each other by sending and receiving events. For example, updates to the display are managed by a *display tool*. Programs that need to be notified when all or part of a scene changes do so by expressing an interest in an *update event*. Each time the state of the scene is updated an update event is broadcast and any tools that have expressed interest will be notified.

In this section we present some of the tools that have been created on top of the prototype system.

I/O Tool

The input/output tool is used to read and write scenes, models, and cue sheets. The length of a scene, in frames, can also be adjusted through this tool.

Display Tool

The display tool manages display list updates for a scene. These updates are typically delayed so that multiple changes to a model result in only a single display update. Users interact with the display tool to alter the *level of detail* and *complexity* with which a scene's models are presented on the display device. Level of detail is a value that controls which parts of a model are displayed. Each object within a model is assigned a level of detail. This assignment is completely under control of the modeler. Only those objects within a model with a level of detail greater than or equal to the current level of detail are incorporated in the presentation of the model. Complexity is a parameter that controls the *detail* with which graphics primitives are rendered. For calligraphic displays, for example, this controls the number of lines used to draw a primitive. In addition to the interactive facilities just described, the display tool also acts as a ``server'' for the back-face cull and quick rendering facilities.

Camera Tool

The camera tool displays a model on the display device. (See the two cameras that appear on left hand side of Figure 4-1.) Cameras come in two forms: animated and static. A static camera remains fixed in space, while an animated camera may be moved about interactively or through articulated variables. Users normally setup one primary camera that represents the point of view that will be used for filming and bring in additional cameras when debugging motion. The camera tool provides an interface to two facilities that are implemented in the display tool: the ability to generate an image with back-facing sides culled and the ability to generate a quick and simple rendering of the image using the Pixar Image Computer (PIC).

Many programs *attach* themselves to a camera in order to provide visual feedback. For example, Figure 6-1 shows a grease pencil tool that can be used to annotate a frame. This tool operates by temporarily taking over control of input events in the area in which the camera tool places its display.

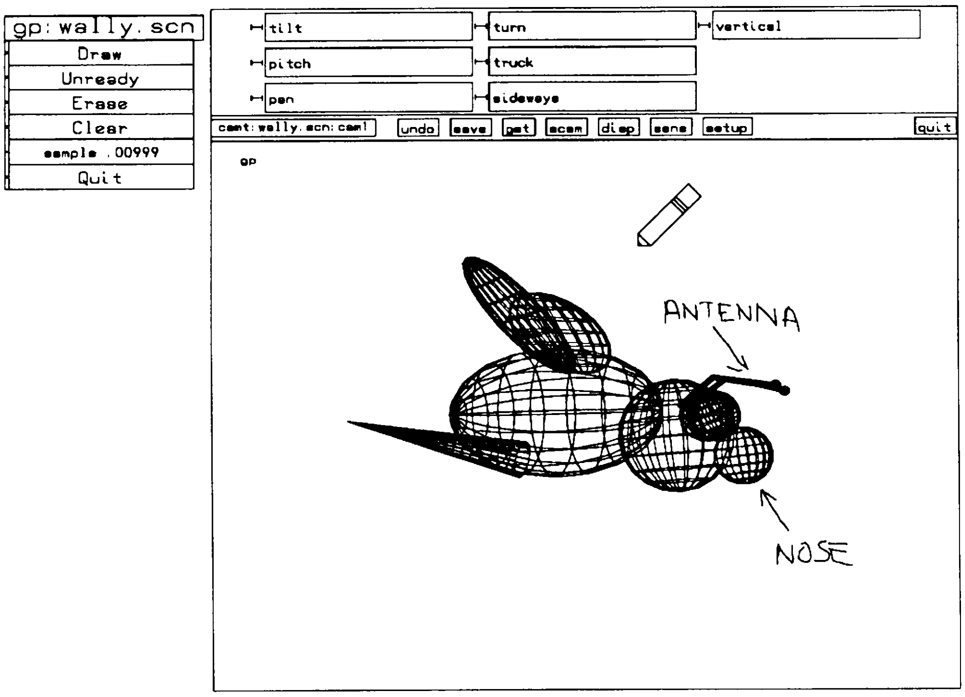


Figure 6-1. Grease pencil tool.

Calculator Tool

The calculator tool provides users with a simple numerical entry facility. The calculator is integrated with other tools through the selection service. Useful numbers that are defined in other tools can be saved in one of the calculator *registers*.

Articulated Variable Spline Editor

The spline tool is an interactive editor for articulated variables, where the variables are edited according to their underlying splines. The spline editing facilities include adding and deleting knots, and altering the value, tension and bias (as applicable) at each knot. Splines are displayed as value versus time within a scene; see Figure 6-2. This presentation can be limited to a specific period of time and/or range of values. Multiple splines/variables can be displayed and edited simultaneously.

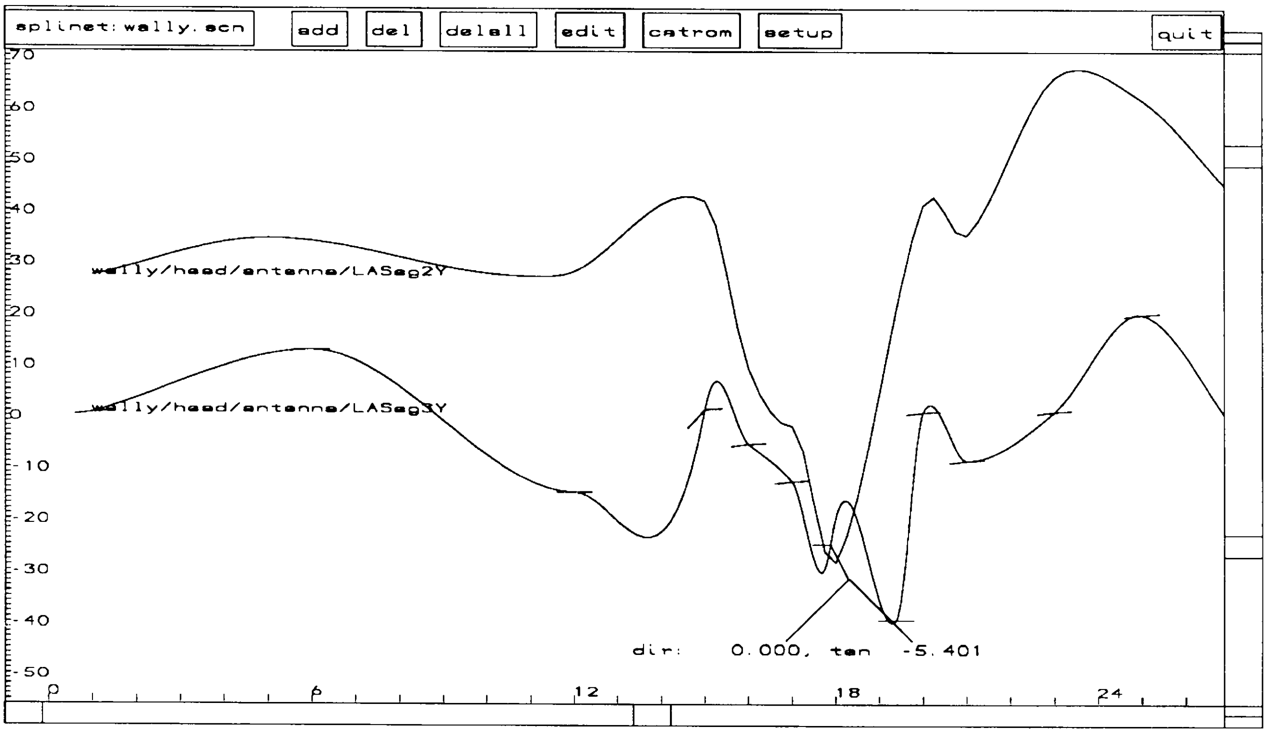


Figure 6-2. Spline editor tool.

Model Language Viewer

The model language viewer displays an ASCII version of a scene's model. The model's internal form is "reverse compiled" and the result is displayed in a window. Text can be selected from the viewer for use with other tools. Future plans call for developing this tool so that it supports editing functions with immediate graphical feedback.

Object and Articulated Variable Selection Tool

The object tool displays the names of objects and articulated variables in a scene. This display is used in conjunction with the selection service to identify objects and articulated variables that are to be manipulated by other tools. For example, to edit the values of an articulated variable as a spline, the name of the variable is selected within the object tool, and then the "add" operation is invoked in the spline editor to add the variable to the set of splines being edited.

Playback Tool

The playback tool implements real-time playback facilities. A range of time is specified over which to play back the motion of a scene through each of the cameras. Time is then cycled and the model is executed to force the display to be updated. For playback on the PS350 display, one cycle is executed and, at each frame, the 2-D vector data in the PS350 refresh buffer is recorded. After the cycle has been recorded as 2-D vectors, the playback tool then turns off the PS350 graphics pipeline and transmits the refresh buffers directly to the PS350 display subsystem. By avoiding the PS350 graphics pipeline extended real-time playback of complex images is possible.

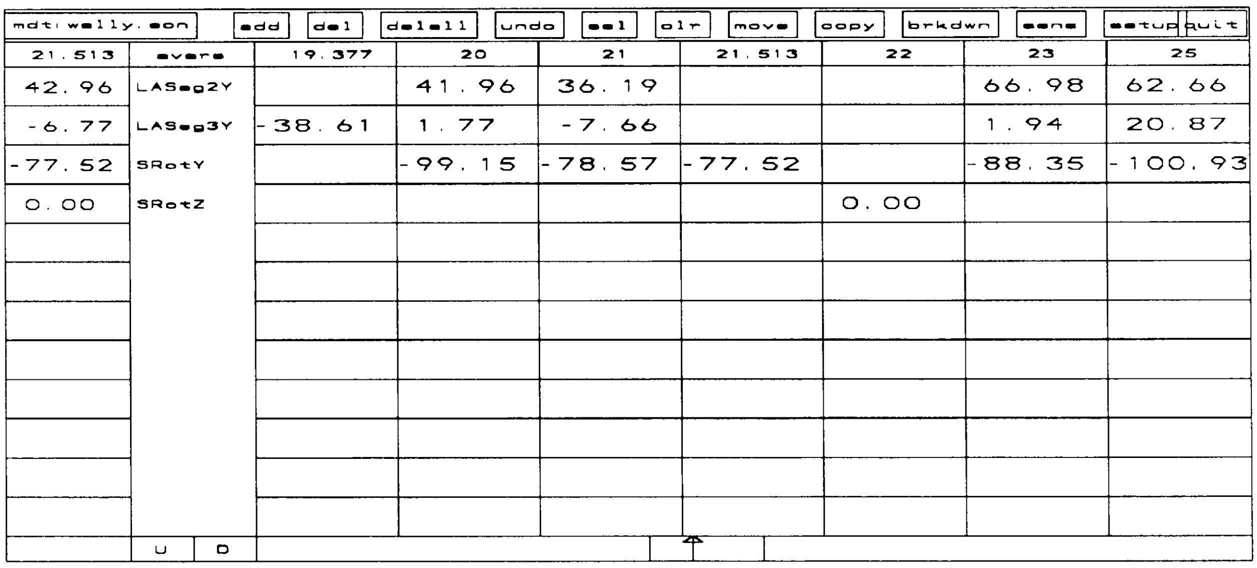
An alternative scheme is also supported whereby a quick render is performed for each frame in a sequence. These frames can then be assembled in the framebuffer of a PIC and the PIC's video controller can be manipulated to provide real-time playback of the rendered images for the scene.

Picking Tool

The picking tool is used to interactively select graphical objects that are displayed in a camera's window. Users can point at a location on the screen and the pick tool will then map this location to an object in the model. The mapping process is performed partially in hardware, using the picking facilities of the PS350, and partially in software. A pick can be resolved not only to an object, but also to a component of an object; e.g. a point in a patch. Picking is performed according to a pick box whose size is under user control.

Cue Sheet Editor

The cue sheet editor is used to edit a scene's cue sheet. The cue sheet is presented as a table with each row associated with an articulated variable, and each column with a keyframe; see Figure 6-3. Users can interactively add, delete, and copy articulated variables and keyframes. The keyframe values for articulated variables can be set either by poking values into a entry in the table, or by manipulating sliders bound to each location in the cue sheet. Values for poking are typically obtained from the calculator.


mdt: wally.scn | add | del | delall | undo | sel | clr | move | copy | brkdwn | scene | setup | quit

21.513	avars	19.377	20	21	21.513	22	23	25
42.96	LASeg2Y		41.96	36.19			66.98	62.66
-6.77	LASeg3Y	-38.61	1.77	-7.66			1.94	20.87
-77.52	SRotY		-99.15	-78.57	-77.52		-88.35	-100.93
0.00	SRotZ					0.00		

U D

Figure 6-3. Cue sheet editor tool.

7. Summary and Future Work

We have described a tool-based 3-D modeling and animation system. This system has been designed to serve us both as a production system for the creation of computer generated animation, and as a testbed for future research. The use of simple tools and a shared workspace permits important flexibility in the system's use and in the incorporation of new facilities.

We are currently working on using the system in the production of an animated film. This work will exercise the extensibility of the system through the addition of new tools required for the production. Future work is expected to focus on the better integration of the system with rendering facilities being developed at Pixar, and with the incorporation of dynamics and physical simulation into the animation facilities.

References

1. Duff, Tom, *md – The Motion Doctor Keyframe Animation System*, Lucasfilm, Ltd., 1984. unwritten technical memorandum
2. Reeves, Bill, *me – A Model Editor*, Lucasfilm, Ltd., 1984. unwritten technical memorandum
3. Leffler, Samuel J., *PS350 Host Interface*, Lucasfilm Ltd., November 1985. original design note
4. Johnson, Neil, "VMEbus/Ibus/Commonbus (VIC) Interface Functional Specification," IAS-Mkg-FS-001, Evans & Sutherland Corp., January 1988. internal specification
5. Sun Microsystems, *NeWS Technical Overview*, Sun Microsystems, Inc., 1987.
6. Reeves, Bill and Sam Leffler, "CPAC2: A Canvas-Based Calligraphic Graphics Package," Technical Memo #149, Pixar, San Rafael, CA 94901, 1987.

7. Sun Microsystems, *NeWS 1.1 Manual*, pp. 19-27, Sun Microsystems, Inc., Mountain View, CA 94903, October, 1987.

8. Leffler, Sam, *A Client Library for NeWS*, Pixar, San, Rafael, CA 94901, January, 1988.

The JUNET Environment

Jun Murai

Computer Centre
University of Tokyo
Japan
jun@u-tokyo.junet

ABSTRACT

The JUNET environment consists of various Kanjified public domain utilities and some original tools providing Kanji capabilities. The design and implementation of this environment will be described, as well as the current status of JUNET itself.

(A paper was not submitted in time for inclusion in the proceedings.)

Measuring File System Activity in the UNIX System

Maurice J. Bach†
Ron Gomes

AT&T Information Systems
190 River Road, Summit, NJ 07901

ABSTRACT

We describe an analysis of system call activity (particularly file system activity) made on several UNIX systems in a software development lab. The measurements were motivated by design work in support of distributed file systems; the intent was to characterize such things as system call frequencies, file system access patterns, and caching behaviour, and to identify performance bottlenecks which might have been missed by existing measurement tools. Among the more important results of the study:

- Most system call activity is file system activity.
- Most *read*s and *write*s are not matched to the file system block size.
- Most terminal I/O is done a single character at a time
- Operations on directories dominate buffer cache activity even though only a small part of the cache contains directory data.
- Most buffer cache hits result from repeated access by a single process.

1. Overview

We describe the analysis of measurements of system call activity taken on several UNIX systems running in out development lab, focusing on file system activity. The measurements were motivated by design work in support of distributed file systems which raised several fundamental questions to which no clear answers were available.

- What are the relative frequencies of system calls? Current report mechanisms give rates for a few system calls (*read*, *write*, *fork*, and *exec* were thought to be high runners) but no numbers were available for all system calls.
- What caching strategies are appropriate for distributed file systems? What data should be cached? What data appear in the cache of single-processor systems? Can any important benefit be derived from the caching of file name data?

Existing performance measurement tools were unable to answer these questions. A profiler reports the relative execution times of operating system routines. Weinberger [4] presents a scheme whereby counters are inserted into all basic blocks of code and the numbers later tabulated. Both schemes give an idea of what the system is doing but not why it is doing it. The kernel routine *getblk*, for example, allocates a buffer from the buffer cache for a particular disk block. The available tools report how many times *getblk* is called but do not say whether the caller intended to read data from a directory, to *write* a regular file, to *exec* a program, to read file mapping information, etc. Another tool, the System Activity Reporter (SAR), gives command summaries and selcted counts but its output is not detailed enough for our purposes.

This study covers system calls in general, *read* and *write* calls in particular, file sizes, evaluation of file names, and use of the disk buffer cache.

† Author's current address: IBM Israel Scientific Center, Technion City, Haifa, Israel.

2. Method

Measurements were made on three 3B20's and one VAX 11/780 in our lab while running their normal loads; generally the number of users per machine ranged between 10 and 35. All of the machines were used primarily for software development. We chose to measure time-shared rather than single-user systems in the belief that a more balanced picture of resource usage would result from a mix of users. The results are thus applicable to a software development environment but may not apply to other environments (office automation, factory automation, real-time processing, or engineering workstations).

Counters were inserted at strategic locations in the operating system code. The overhead of this additional code was less then 5% of system throughput as measured by a standard benchmark. Periodically user-level programs read the counters, dumped them to a file, and re-initialized them.

Three data samples were collected on each machine every working day for six months. Data samples were accumulated from 8 AM until 1 PM, from 1 PM until 6 PM, and from 6 PM until 3 AM, but the third set of results was inconsistent with the others and was not considered in the study. We thus assume (not unreasonably) that activity between 8 AM and 6 PM of a work day is "typical" of time-shared UNIX systems and activity at other times is not. Data collection was cumulative so there is no recognition of spurts of activity.

All systems were running System V Release 2. No attempt was made to analyze other UNIX systems, such as System V Release 3 or 4BSD.

3. System Calls

These measurements were straightforward; at every occurence of a system call an associated counter was incremented and the values of the counters later compared to yield the relative frequencies of system calls.

system calls	% of all calls
read	30-40
write	10-20
alarm	6-13
signal	6-10
lseek	4-6
close	3-5
ioctl	2-3
open	1-2
stat	1-2

Table 1. Frequencies of system calls.

Table 1 shows the frequencies of the most commonly used system calls. *read* and *write* together account for 50-60% of all system calls but the ratio of the number of *reads* to *writes* may vary between 1.5 and 4. Calls related to file system activity comprise 75-90% of the total. *fork* (create a new process) and *exec* (run a new program) together account for less than 1% of all system calls, too low to be listed in the table. *lseek* (seek to a given offset within a file) occurs more frequently than expected, probably because text editors use it to manoeuver through edited files. Many commands use *signal* (handle a software interrupt) to handle asynchronous events gracefully so the high incidence of *signal* is not surprising. However, the number of calls to *alarm* (schedule a timeout) seemed much too high. Some network protocol code runing at user level on these systems uses *alarm* and *signal* to implement timeouts but commands that exercise the network were not used enough to account for this. Further investigation revealed that the *vi* text editor on System V uses *alarm* to interpret escape sequences arriving from the keyboard: if such a sequence arrived fast enough it is interpreted one way, otherwise it is interpreted another way. *vi* is heavily used in our environment and this could explain the high incidence of *alarm*.

One other seemingly anomalous result merits explanation: *close* occurs more than twice as often as *open* (even though they might be expected to occur in pairs) because when a process is created it inherits a number of already-open files.

The results suggest, not surprisingly, that improving file system performance can have a dramatic impact on overall system performance; this has been demonstrated by others [2]. It could be claimed that *fork*, *exit*, and *exec* consume a lot of CPU time compared to *read* and *write* and that the emphasis given here to file system calls is thus inflated. But *fork* in System V is implemented using copy-on-write so its overhead is low. *exit* incurs the overhead of freeing memory for a process, but this is also low in a demand paging environment. Finally, most executable files on System V now use a file-mapping scheme to demand-page executable files instead of loading the file into memory on invocation. *exec* thus interacts with the file system, so improvements to file system performance should have a direct positive effect on *exec* and on overall performance.

4. Read and Write

Special attention was given to the analysis of *read* and *write* because of their high frequency. There are two parts to the analysis: measuring the frequency of calls for particular file types and measuring the frequency distribution of return values for particular calls. (The return value from *read* or *write* is the number of bytes written.)

file type	% of all reads	% of all writes
pipe	5	4
device	35-60	55-80
regular file and directory	35-60	20-45

Table 2. Frequency of read/write calls per file type.

4.1. Regular Files, Directories and Pipes

Table 2 shows the frequency of *read* and *write* calls for different file types, including regular files and directories, pipes (inter-process communication channels), and device files. Access frequencies for regular files, directories, and devices vary widely across samples. From the data it is difficult to generalize about the ratio of *reads* on devices to *reads* on regular files and directories, but processes generally *write* devices more than they *write* regular files (a ratio of 2 to 1 is typical); it is likely that terminal I/O accounts for much of this. The frequency of pipe use is low (although pipes have great impact both on system functionality and on users' perception of it). We can conclude from this that pipe throughput rates are insignificant to overall system response time and throughput, assuming the amount of data flow through pipes is small (as we show below). Processes that use pipes usually operate in pairs (reader and writer) and thus already incur the added expense of context switches, further mitigating the perceived cost of sending data through a pipe.

The return values from *read* and *write* system calls were tallied in order to measure use of the file system logical block size in such calls. The return values were categorized into ranges chosen intuitively and are summarised here.

Return values from *read* were startling. Depending on machine and sample, only 30-50% of all *reads* of regular files return 1024 bytes of data (a file system block) and 35-50% return between 8 and 64 bytes. Return values from *reads* of directories are even more striking: over 75% (and sometimes over 90%) of all such calls ask for and return between 8 and 64 bytes. Each directory entry in System V is 16 bytes long, and many programs *read* one directory entry at a time‡. Clearly many programs do not use the standard I/O package, which gears the size of its *reads* to the size of file system blocks. ''Nibbling'' files results in frequent *reads*, extra system call overhead, and reduced throughput. Similarly only about 25% of all *writes* are done in units of the file system block size. It is likely that performance tuning of some commands can improve overall performance without the need to touch the operating system at all.

Similar results were expected for pipes since they are typically used to pass data between programs which, if ''well-written'', have no foreknowledge of the file types with which they will be used. But about 30% of all *reads* and *writes* on pipes transferred between 2 and 7 bytes if data, suggesting that they may be aware of fixed length records transmitted through pipes. Sets of processes may use pipes to synchronize execution, transmitting small amounts of information through the pipe.

4.2. Devices

Terminals were by far the most frequently accessed devices (receiving 83% of all device *read* requests and 96% of all device *writes*). Very little use of tape or raw disk was detected. A surprisingly high percentage of all device *reads* (14%) were for the physical memory device, probably issued by programs such as *ps*

‡ This has changed in System V Release 3, which incorporates a new system call for reading directories.

and *sysmon* which read system internal data structures directly to report on system activity and resource usage.

Only about 3% of all I/O calls on these systems made use of the high-speed local area network (LAN). Although the number of *read*s and *write*s issued for terminals dwarfs the number of such calls for the network, the amount of data movement is much higher for the network: the total number of bytes transferred favours the LAN over terminals by a factor of 2 or 3 to 1*. We conclude that in this environment network device drivers should be optimized for rapid movement of large amounts of data.

Single byte *read*s and *write*s account for about 80% of all such calls on terminal devices. This result is not too surprising since heavily-used programs such as the screen editor *vi* read, interpret, and echo characters one at a time as they are typed. Multi-byte I/O would likely be faster but the instant-response paradigm of text-editing (using ''raw I/O'') largely precludes this. Clearly one-byte manipulation is extremely important for fast terminal response, and terminal hardware and software drivers should be optimized for this case.

5. File Sizes

Mullender and Tanenbaum [3] report that 85% of all files on their UNIX systems are smaller than 8KB and 48% are smaller than 1KB, but their study didn't measure actual (dynamic) file use. The measurements reported here record the sizes of regular files and directories at *open* time and the sizes of *exec*ed files at the time of *exec*. The numerical ranges of file sizes (multiples of 4KB) were chosen to simplify the calculations and so that data would not be lost at the high end. As it turned out, more attention should have been paid to the low end.

The measurements for regular files support the results of Mullender and Tanenbaum. At least 76% of all open, regular files are under 10KB (the structure of the file system makes such files cheaper to access). Of the 75% that are smaller than 4KB a large number are probably smaller than 1KB, although the granularity of the measurements does not show this. File sizes between 24KB and 64KB appeared to be uniformly distributed.

Similar results hold for directories: 82% of them were smaller than 4KB. However, 18% of all directories directly *read* by user programs were between 4KB and 12KB in length, a disturbing result. A search for directories larger than 4KB on one machine turned up several dozen, many of them ''spool'' directories of some sort. Many entries in these directories are empty, but the directory size is huge and the cost of searching them high because the system does not perform compaction of directories.

The results suggest that file access strategies that concentrate on fast access for very small files (under 4KB) and moderately small files (under 24KB) would yield good results.

The distribution of sizes of *exec*ed files tends to grow more slowly than the distribution of sizes for *open*ed files. More than half of all *exec*ed files are smaller than 20KB. 94% of all *exec*ed files are smaller than 64KB. When demand paging is done directly from the executable file, the system constructs a linear list of disk block numbers to avoid the expense of indirect block accesses when a process incurs a page fault. The small size of *exec*ed files suggests that in general the block list consumes very little memory for the performance gain it provides.

It would be interesting to extend this analysis to compare virtual memory sizes as processes execute. It might be expected that many programs have a small amount of text but large data requirements that are not reflected in the size of the executable file.

* At the time of this study network functionality was primitive on the sampled systems compared to services available on other systems (System V Release 3 and 4.2BSD for example). It would be interesting to investigate the effect of such features as a distributed file system and remote login on network use.

6. File Name Lookup

namei is the kernel routine that parses a file name and finds its *inode*, the system's internal representation of a file. *namei* is called on behalf of such system calls as *open*, *stat* (get the attributes of a file) and *creat* (create a new file or truncate an old one). Counts were kept of the number of times *namei* was called on behalf of each system call.

caller	% of all *namei* calls
stat	30
open	25
access	15
unlink	9
exec	7
other	14

Table 3. Frequency of *namei* calls.

About 6-10% of all system calls invoke *namei*. More than half of the calls to *namei* were on behalf of *stat* and *open* (see Table 3). Although the system calls that use file names are made as frequently as *read* and *write*, *namei* reads directories internally and has inner loops that make heavy use of the file system.

6.1. Number of File Name Components

The file system has a hierarchical structure in which a file is named by a ''path'' of several components identifying the directory structure in which it is contained. *namei* loops for each component of a path so measuring the number of components gives an idea of how much work it does. The average number of components per *namei* call is 3.2. Users at a terminal probably use file names with fewer components because they typically change directory to reduce typing. It is not known which programs generate long file names (except possibly the command interpreter when searching command directories). Further study in this area may uncover interesting situations.

6.2. Length of directory Searches

Each file system block contains 64 directory entries. About 77% of all directory searches found the target entry in the first block of the directory and 5-10% found the entry in the second block. 18-23% of directory searches read more than 2 blocks (2KB) (consistent with our earlier result that 18% of all open directories have a size between 4KB and 12KB). The number of blocks read during a directory search averages between 1 and 1.2 per component. Given that at least 1 block must be read, these numbers look good. But even assuming an average of 1.1 blocks per directory this means that 10% of all directories searched have more than 64 entries, which seems excessive. On the measured systems some commonly used directories such as ''/bin'' and ''/usr/bin'' were found to contain between 100 and 200 entries (2 to 4 blocks worth of data).

Several performance improvements are suggested. Reduction of the number of multi-block searches could improve system performance. The system does not sort file names in a directory, so their location remains constant after creation. It would be easy to determine the most frequently used commands in common directories by inspection of standard accounting files, and administrative programs could periodically sort the entries in a directory to achieve better search times. Alternatively the system could sort the directory automatically or perform compaction (possibly incurring more overhead in the name search). Another possibility would be to vary the linear search order currently used. Finally, use of a name cache could reduce the amount of CPU time as well as the amount of disk I/O required for directory searches (though measurements of the buffer cache, described below, indicate that a substantial amount of I/O caching benefit is already in place for directory data). All of these proposals merit investigation.

6.3. Inode Caching

The system maintains a cache of currently and recently referenced inodes in an attempt to reduce both disk traffic and use of the buffer cache (analyzed separately below). Between 54 and 62% of all referenced inodes are found to be already in the inode cache with a reference count of 1 or more (meaning they are currently in use), another 31-40% are found in the cache with a reference count of 0 (meaning they were recently in use), and 5-9% are not found in the cache at all. In short the inode cache enjoys a hit ratio of 90% and is effective in ''protecting'' the buffer cache; buffers that contain inodes can migrate out of the buffer cache and allow other buffers to take their place. The high cache hit ratio indicates high locality in file references. Inodes tended to incur between 5 and 13 hits while in the cache. The cache size on the measured systems was between 400 and 600 entries: this was not varied for this study so as not to change the production environment too radically.

operation	% of all buffer use	% of all hits	% hits/freq of op
namei	25-35	25-40	90+
read regular files	10-20	10	75-85
reading directories	10-15	10	90+
bmap (read)	5-8	10	90+
page faults	5-8		40-50
stat	5		90+
iupdate	5		90+
iread	2-3	5-10	45-60
write	5-7	10-12	90+
alloc			20-30
read pipe			90+
write pipe			90+
other		0-30	

Table 4. Frequency of buffer cache usage.

7. Buffer Cache

The system maintains an internal file system buffer cache, keyed on logical block number, to reduce the number of disk accesses. The four measured systems were configured with 400, 900, 1000 and 1100 buffers. (Again, the sizes were unchanged for this study.) Table 4 shows the frequency of buffer cache access on behalf of different file system operations:

- *namei* refers to buffer cache operations performed while parsing file names.
- *read regular files* refers to operations for reading file data blocks.
- *reading directories* refers to operations while reading directory data blocks with the *read* system call.
- *bmap*: cache operations done while reading indirect blocks (which contain the disk addresses of file data blocks).
- *page faults*: operations done while handling page faults.
- *stat*: operations to extract the access/modification time of a file from an inode.
- *iupdate*: operations to write the contents of an inode to disk.
- *iread*: operations to read the contents of an inode from disk.
- *write*: operations for writing data to files.
- *alloc*: operations to find free blocks in the file system.
- *read pipe* and *write pipe*: operations for reading data from, or writing it to, a pipe.

Although *read* and *write* occur with greater frequency than system calls which perform name lookups, Table 4 shows that more buffer cache operations result from *namei* because of its inner loops (several components per file name and several directory blocks per component). In particular, *namei* is responsible for most directory accesses. Regrouping operations into broader categories, 30-35% of all buffer cache operations are caused by *namei*, 25-30% are caused by file operations (*read*, *write*, *bmap*). 10-15% by directory *read*s from user programs, 12% by inode operations, and 10-20% by miscellaneous operations. These numbers emphasize the particular importance of directory searches to good performance — between 40% and 50% of all buffer cache activity is for directories.

7.1. Cache Hits

The second column of Table 4 shows the distribution over all operations (for example, 25-40% of all cache hits in the system are caused by *namei*), and the third column shows the percentage of hits incurred by each operation (over 90% of all *namei* operations induce a cache hit). Blank entries indicate negligible contribution to the total number of cache hits. Comparison of the first two columns shows that *namei* is the most frequent user of the cache and is responsible for a proportionate number of cache hits. Reading files and directories (*read regular*, *read directory*, *bmap*) accounts for about 25-40% of all cache activity and a similar proportion of cache hits. Writing files accounts for fewer cache operations but a proportionately larger number of cache hits.

The third column of Table 4 shows that buffer cache hit ratios are extremely high, with many operations enjoying a 90% cache hit ratio or higher. In particular directory operations (*namei* and reading a directory) have a better than 90% hit ratio, indicating that most requested directory data is accessible without the need for disk I/O. A separate component name cache has been shown to improve performance on other systems [1] but since a substantial caching benefit is already derived from the buffer cache the additional gain achievable from a name cache may be less than expected.

The cache hit percentage for reading regular files was in the 75-85% range, still surprisingly high. A lower hit ratio was expected here; the presumption is that users *read* many different files without accessing the same data repeatedly. But since many *read*s access only a few bytes at a time successive calls are likely to access the same disk block and produce cache hits on the associated buffer. (If more programs issued large *read*s we might expect the hit ratio to drop.)

Cache hit percentages for the allocation of a new block in a file were in the 20-30% range. A high frequency was not expected here because such blocks are being used for the first time in a file. Nevertheless there are some cache hits on buffers whose associated blocks may have been recently freed from a file which was removed; temporary files might be expected to encounter this, suggesting that the LIFO policy currently used by the system for free block allocation is appropriate. Inode operations in *stat* and *iupdate* also have better than a 90% hit ratio because they occur soon after other operations that use the inode, making it likely that the inode will still be in the buffer cache.

Cache hit percentages for *iread* were in the 45-60% range: if a file is used often its inode will usually be in the inode cache so its disk inode will rarely be accessed. Many of these buffer cache hits may result from references to other inodes which happen to reside in the same disk block. Cache hit percentages for handling page faults were in the 40-50% range. Because of the discrepancy between page size (2KB) and file system block size (1KB) on the 3B20, block read-ahead is used whenever there is a page fault. After the system successfully reads the first block of a page, it probably takes a cache hit for the second block. On the VAX, where page size is 512 bytes, 2 pages are read when a page fault occurs. there is a good chance that a process will fault on both pages if the virtual addresses are close to each other, thus causing a buffer cache hit on the second page fault.

7.2. Cache Hits by the Same Process

Many cache hits result from repeated access by a process without intervening hits by other processes. More than 80% of the cache hits caused by reading regular files and directories, writing regular files, and executing *bmap* operations result from a previous access by the same process. Frequently the ratio approaches 100%: when a file is accessed by a process there is not usually much concurrent use of that file by other processes. This is not unexpected in a development environment in which most users access private files even though there is a large pool of public files that are frequently accessed by all users. Results might be different in other situations (such as a transaction environment) in which there may be more access to common data. The implication of the high repeated-hit rate for a process is that the system could profitably flush data buffers when a process *close*s a file — an important consideration for data caching in a distributed file system.

75% to 80% of all buffer cache hits caused by *namei* (cache hits on directory data) are repeated hits by the same process; this is higher than expected and indicates that directories tend to be accessed repeatedly by one process without any intervening access by other processes. There are several ways in which this can happen. A command such as *ls*, which lists the names and attributes of the files in a directory, will first *read* the target directory (bringing it into the buffer cache) to find the names of the files it contains and will then issue a *stat* system call for each file in order to get its attributes; *stat* invokes *namei* and will elicit buffer cache hits. And the *shell* (command interpreter), for every command typed, searches a set of command directories and might be expected to hit the same directories repeatedly. These two examples alone are insufficient to explain the high repeated hit rate for *namei* (neither operation occurs frequently enough). But one implication of the high hit rate is that it is reasonable to optimize the directory search algorithm based on the predicted behaviour of a single process without being too concerned about the effect of intermingled references by other processes.

Two other results: page faults exhibited repeated buffer cache hits (80-90%) despite the fact that the system maintains a separate page cache and it would be expected that pages would migrate out of the buffer cache. And I/O on pipes elicited about 40% repeated hits with *write*s consistently producing more repeated hits than *read*s; the lower repeated hit rate for pipes can be explained by the interaction between reader and writer processes which share and interleave accesses to cache buffers.

7.3. Composition of the Buffer Cache

The number of cache misses was counted for every operation. When a buffer was reassigned as a result of a cache miss it was tagged with the type of the operation that caused it to be in the cache. (Thus it is possible to speak of ''*namei* buffers,'' ''*iread* buffers,'' etc.) When a buffer migrated out of the cache to make room for an incoming block the tag was noted and an associated counter incremented. The two sets of frequencies — cache misses caused by particular operations, and buffer types leaving the buffer cache — were equal. This implies that the buffer cache achieves a steady state that can be divided between a fixed set of buffers that never leave the cache and a more volatile set that move in and out. Periodic sampling found few blocks that remained constantly in the cache; effectively the volatile set of buffers takes up the entire buffer cache.

The composition of the buffer cache is remarkably consistent. Directories occupy 4% of the cache, inodes occupy 5-10%, data blocks (read and write) occupy about 40%, and pages of virtual memory occupy about 35%. Comparison of these numbers to those in Table 4 reveals an inverse relationship: operations such as *namei* that are used most frequently and that have high hit ratios occupy few buffers in the cache while operations such as page faults that are infrequent have many buffers of the corresponding type in the cache.

The number of hits accumulated by each buffer type was measured and the distribution of hits across buffer types was approximately the same as the distribution of hits across buffer operations (Table 4), as expected. In addition the number of hits taken by individual buffers was measured; a buffer typically incurred between 6 and 11 hits while in the cache.

Measurements were also made of the number of times each buffer was written while in the cache. These showed that frequently-accessed buffers, in particular buffers that enter the cache because of *read*s on regular files or directories, *namei*, page faults, and *bmap*, are rarely, if ever, written.

8. Conclusions and Suggestions for Future Work

We summarize the major conclusions of this work. Some of the following points may seem truisms but bear repeating. They apply to a program-development environment but may not extend to other environments. There may also be a three-blind-mice-and-the-elephant syndrome caused by looking at file system activity without taking process scheduling and memory management into account. However, the techniques employed in this work can and should be used to measure the characteristics of system execution in other environments and in other parts of the system kernel, particularly process scheduling, memory management, and remote file sharing.

- Most system calls deal with the file system. Indeed, most system calls are *read* and *write*.
- The ratio of the number of *read*s of regular files and directories to the number of *read*s of devices varies from system to system according to workload. However, devices (especially terminals) are written more frequently than regular files.
- Most *read*s and *write*s do not match the data size to the file system block size (nor use the standard I/O library). Performance would be improved if they did.
- Few *read*s and *write*s deal with pipes. Pipe throughput is insignificant to overall system throughput and response time.
- Most device *read*s and *write*s deal with terminals. There is a lot of one-byte terminal traffic; terminal drivers and hardware should be optimized for this case and programs should avoid one-byte data transfer because of the system overhead it generates.
- In the environment studied most network traffic requires the transfer of large blocks of data; device drivers should be optimized for this case. Network traffic dominates terminal traffic in the number of bytes sent, but the number of I/O system calls on terminals dominates the number of such calls on network devices.

- 76% of all regular files are smaller than 4KB and 96% are smaller than 24KB. The file system should be optimized for these cases.
- Most directories are smaller than 2KB but a few are as large as 12KB. Many large directories have many empty slots; their size could be compressed to speed up directory search.
- 94% of all executable object files are smaller than 60KB; 53% are smaller than 20KB. Demand paging algorithms may be able to take advantage of the small text size.
- File names used in system calls have about 3 components on average. Most component names are found quickly (in the first block of a directory).
- The inode cache has a hit ratio of better than 90%. Most system calls that access a file by name access a file that is already ''in use''.
- Buffer cache hit ratios are close to 100% for directory search operations and between 75% and 85% for *read*s of regular files. A separate component name cache would probably improve performance by reducing the amount of computation required by a name search. It would also help reduce the amount of disk I/O, although the high buffer cache hit ratio for directories indicates that a high percentage of requested directory data is already available without the need for disk I/O.
- The buffer cache is used more for directory searching (particularly in the *namei* function) than for any other activity. Activity for reading files (data blocks, indirect blocks) is a distant second. However, very few buffers in the cache contain directory data. A relatively large number are devoted to regular file data and virtual memory.
- Although many processes frequently access the same files, their references to the file are localized; processes frequently produce repeated cache hits in accessing a file without intermittent hits by other processes.
- Buffers that are frequently used are rarely written, particularly directory blocks and regular file data blocks.

These results suggest a number of areas for future performance work.

- A design for caching data in remote file systems has already been implemented based on this work. The benefits of extending the caching scheme to include remote file name caching should be explored.
- Frequently-used commands should be recoded to reduce the number of *read*s and *write*s they issue. In many cases effective use of the standard I/O library would suffice. Other commands should be rewritten to make better use of system resources.
- The device subsystem should be examined carefully for performance bottlenecks, particularly the terminal subsystem and network protocol and driver code. This examination should extend to hardware and firmware.
- New statistics gathering counters should be placed in the system to count all system calls, file type accesses, etc. Measurement programs should be available to report these statistics and in particular it should be possible for administrators to compare local system activity to remote file sharing activity.
- Processing scheduling and the memory management subsystem should be examined using methods similar to those described in this paper.

9. Acknowledgements

Much of the analysis presented here grew out of discussion with Steve Buroff, Gerry Gordon, Tom Houghton, Mark Luppi, Anna Melamed, Art Sabsevitz, Curt Schimmel, Larry Wehr, and probably others. Mark Luppi and Kang Yueh used the results as the basis for the implementation of a data caching scheme for remote file systems. Steve Bellovin, Brian Kernighan, and Y. T. Wang reviewed earlier drafts of the paper. Roy Feigenbaum and Grace Wang provided access to the kernels of the sample machines.

10. References

1. Leffler, Sam *et. al.*, ''Measuring and Improving the Performance of 4.2BSD,'' Proceedings of the USENIX Association Summer Conference, Salt Lake City, June 1984, pp. 237-252.
2. McKusick, M.K. *et. al.*, ''A Fast File System for UNIX,'' *ACM Transactions on Computer Systems*, **2**(3), August 1984, pp 181-197.

3. Mullender, S.J. and A.S. Tanenbaum, ''Immediate Files,'' *Software — Practice and Experience*, **14**(4), April 1984, pp. 365-368.

4. Weinberger, P.J., ''Cheap Dynamic Instruction Counting,'' *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 6, Part 2, October 1984, pp. 1815-1826.

Yacc Meets C++

Stephen C. Johnson

Ardent Computer Corp.†
880 W. Maude Ave.
Sunnyvale, CA, USA, 94086

ABSTRACT

The fundamental notion of attribute grammars [1] is that values are associated with the components of a grammar rule; these values may be computed by *synthesizing* the values of the left component from those of the right components, or *inheriting* the values of the right components from those of the left component.

The Yacc parser generator, in use for over 15 years, allows attributes to be synthesized; in fact, arbitrary segments of code can be executed as parsing takes place. For the last decade, Yacc has supported arbitrary data types as synthesized values and performed type checking on these synthesized values. It is natural to think of this synthesis as associating a value of a particular type to a grammar symbol when a grammar rule deriving that symbol is recognized.

Languages such as C++ support abstract data types that permit functions as well as values to be associated with objects of a given type. In this framework, it appears natural to extend the idea of computing a value at a grammar rule to that of defining a function at a rule. The definition of the function for a given object of a given type depends on the rule used to construct that object.

In fact, this notion can be used to generalize both inherited and synthesized attributes, unifying them and allowing even more expressive power.

This paper explores these notions, and shows how this rule-based definition of functions allows for easier definitions and much more flexibility in some cases. Several examples are given that are hard to express using traditional techniques, but are naturally expressed using this framework.

Introduction

When we write a grammar rule such as

```
      expr : expr '+' term ;
```

we frequently associate values with the components of the rule and use these values to compute the values of other components. For instance, in the above example we might wish to associate integer values with the symbols *expr* and *term*, and include a rule for generating the value of the left-hand *expr* from the value of the right-hand *expr* and *term*. Values associated with the left side of a rule that are computed from the values on the right are called *synthesized* attributes.

In other cases, we wish to use values associated with the left side of the rule to compute values of components on the right side of the rule. These values are known as *inherited* attributes. One important example of inherited attributes involves passing symbol table information into expressions and statements.

Allowing inherited and synthesized attributes to be specified without restriction makes it difficult to generate parsers automatically from the rule descriptions. For example, it is very difficult even to decide whether the definitions are potentially circular [2]. Nevertheless, attribute grammars are very expressive, making them attractive both as a basis for editors and compilers [3] and as a platform for more extensive computational and database systems [4].

† Much of this work was done when the author was employed by AT&T Information Systems

In a parallel development, languages such as Ada [5], Modula-2 [6], and C++ [7] have explored ways of extending traditional program language type structures in much the same fashion. In Ada and C++, the type concept is extended to include not just the values and the data but also the actions that may be performed on these values. The key idea is that an abstract model of the data type, and the operations provided on it, is presented to the programmer; the details of the implementation are hidden.

Yacc is a parser generator that has been available under UNIX since the early 1970's. The original versions of Yacc permitted only one attribute, of integer type, for each grammar symbol. Later versions allowed other types, including structures, to be computed. However, because the parsing method is bottom up (LALR(1)), and the actions are executed as the rules are recognized, only synthesized attributes can be handled directly. More complex translations must be done by building a parse tree, and then walking this tree doing the desired actions.

In Yacc, a unique datatype may be associated with each nonterminal symbol. In this case, every rule deriving that nonterminal must return a value of the defined type. Each token may also have a defined type; in this case, the values are computed by the lexical analyzer. An action computes the value associated with the left side of a rule; this action depends on the particular rule, and the values of the components on the right of the rule.

In trying to extend Yacc to handle C++, these two streams naturally came together in a prototype tool called y++. Since there was already an association of types with nonterminals, it became natural to ask whether functions could be defined on these types as well, and what meaning this might have. Some examples proved compelling.

Examples

Given a rule

```
expr :  expr '+' expr ;
```

we might choose to define a function *print()*, on the nonterminal/type *expr*. In the context of this particular rule, *print()* might be defined as

```
print()  { $1.print() ; putchar( '+' ) ;  $3.print() ; }
```

(As with Yacc, we will use $1, $2, etc. to refer to the components of the right side of the rule, and $$ to refer to the left side).

We might also choose to define another function, *type*, on *expr*, and this might have a totally different definition:

```
type()  { exprtype( '+' , $1.type() , $3.type() ) ; }
```

Since tokens are only created by the lexical analyzer (and never by rules), functions such as *print* and *expr* can be called implicitly as part of a particular rule, and need no special definition mechanism.

By allowing these rule-defined functions to have arguments and return values, we get many of the effects of inherited attributes:

```
type( TYPE t )  { $1.type( t ) ; $3.type( t ) ; }
```

In C++, a function that is defined as a member of a class can obtain access to the values in an instance of that class; the keyword *this* allows such values to be explicitly manipulated. In y++, when a function *f* is defined on a nonterminal/type *X*, not only the value, but also the function definition itself depends on the rule used to define the instance of *X*.

A nice example is given by the rule:

```
expr :  expr '+' expr ;
```

on which we define two functions, *polish* and *revpolish*:

```
polish()  { putchar( '+' ) ; $1.polish() ; $3.polish() ; }
revpolish()  { $1.revpolish() ; $3.revpolish() ; putchar('+') ; }
```

These two functions can be similarly defined for other expressions. Two input line types can be defined as well:

```
line :   "PRE"  expr
line :   "POST"  expr
```

and a function, *print()*, that is defined as

```
print()   { $2.polish() ; }
```

on the first rule, and

```
print()   { $2.revpolish() ; }
```

on the second. Then, after a *line* has been recognized, *print* will print the expression in either prefix or postfix Polish form, depending on the initial keyword of the line.

The attribute grammar approach to this would require generating and storing both the prefix and postfix translations, and many intermediate translations as well. The above technique saves both space and time.

To summarize this section: we associate datatypes (C++ classes) with nonterminal symbols and tokens. In addition to values, these types have functions associated with them whose definition depends on the rule used to recognize a particular instance of the type. This mechanism generalized both inherited and synthesized attributes; later sections discuss implementation and other applications.

Implementation

We have prototyped a tool, y++, to explore the semantic and syntactic implications of these ideas. Some features of y++ are:

1. Every grammar symbol, token and nonterminal alike, is associated with a C++ class.
2. Every class has 0 or more values, and 0 or more functions, defined on it.
3. Every grammar rule may have associated with it 0 or more functions that may be invoked to access and change the values accessible to that rule. These functions may access and change values of the result (left side) of the rule, and the components (right side) of the rule, and invoke other functions defined on the components of the rule.

In practice, y++ specifications are transformed to C++ programs and compiled. *yyparse* returns a value that is, in effect, a pointer to the parse tree. After calling *yyparse* (which causes some input to be read and parsed), the returned value is used to access the functions that are defined on the start symbol; presumably, these cause transformations and output to be done.

When *yyparse* is called, it creates a data structure that represents the parse tree. For tokens, it creates space large enough to hold the values, if any, included in the token. For nonterminals, the space created depends on the rule used to create the particular instance of the nonterminal. The rule

```
A :  B  C  D  ;
```

would cause space to be allocated as follows:

```
integer rule number
space for the A values
pointer to the B value
pointer to the C value
pointer to the D value
```

In the case of simple actions, not depending on the rule, we simply index past the value to obtain the components. In the case where the actions depend on the rule number, we generate a conditional based on the stored rule number. In the case where B, C, or D is a token, the value returned from the lexical analyzer is saved instead of a pointer.

There are a number of scope issues not yet resolved. If there are two calls to *yyparse*, for example, does the second parse tree overwrite the first, or do both remain active. The issue of default actions and values is also tricky. There is little point in wasting space on characters and literal keywords returned by the lexical analyzer when these are implicitly known from the rule number; the questions is how to recognize this and exploit it.

A similar issue is the treatment of default functions. If a function is called for a rule that contains no definition for that function, should a default definition be assumed? We currently consider this a semantic error and produce a message, *semantic error*, by analogy with the *syntax error* message of Yacc.

Another issue is error handling. There is a premium in being able to return a sensible structure for any input, even those in error, to allow the user to craft special functions that give particularly good error messages. The exact mechanism by which these *error rules* might be constructed is still open.

Finally, given a data structure representing a parse tree, it is very nice to be able to rewrite it; y++ should probably provide such operations through a user interface.

A Simple Example

This section sketches how y++ can be used to make a preprocessor that translates extensions into a base language. We begin with a function *ident*, defined on every nonterminal symbol of the base language grammar; this function, when called, produces a literal text representation of the rule. For example, for the rule:

```
stat :  expr  ';'
```

we might define *ident* as

```
ident()  { $2.ident() ; putchar( ';' ) ; }
```

This grammar can quickly be extended to a preprocessor by simply adding rules for the new constructions, and defining *ident* on the new rules to translate into the base definition. For example, suppose we wish to augment C with the *forever* statement. After recognizing the keyword in the lexical analyzer, we add the rule:

```
stat :  "forever"  stat  ;
```

and the associated definition of *ident*:

```
ident()  { printf( "while(1)" ) ; $2.ident() ; }
```

Clearly, translators that required symbol tables, etc., would be harder, but one could envision a standard C grammar and lexical analyzer being far more reusable in y++ than in Yacc.

Impressive Example

Giegerich and Wilhelm [8] have discussed the difficulty of generating "short-circuit" evaluation of Boolean expressions using the usual forms of syntax-directed translations (See also Aho, *et. al.*, [9]). This becomes relatively straightforward in y++. A function, *bool_gen(t, f, n)*, is defined on the rules involving the short-circuited operators. *t* is the label to go to if the expression is true, *f* the label for false, and *n* has the "preferred" label, either *t* or *f*. The rule

```
expr :  expr  OR  expr
```

for example would define *bool_gen* as

```
bool_gen( t, f, n )
{
        int x = get_new_label();
        $1.bool_gen( t, x, x );
        define_label( x );
        $3.bool_gen( t, f, n );
}
```

and similarly for AND and NOT. The definition for those expressions not involving short-circuit operations would look like:

```
bool_gen( t, f, n )
{
        . . . /* get the value of the expression $1 */
        if( t == n )
                { . . . /* branch if false to label 'f' */ }
        else
                { . . . /* branch if true to label 't' */ }
}
```

Conclusion

This paper describes a simple extension of parser generators to handle abstract data types; in this way, some translations can be specified easily that would be more difficult to describe with conventional attribute grammars.

Moreover, the notions seem to generalize attribute grammars, while at the same time allowing the ideas of Yacc to be brought to bear on the concepts in C++, or perhaps *vice versa*.

References

1. Knuth, D. E., Semantics of context-free languages, *Math. Syst. Theory*, **5**, No. 1, *pp*. 95-96, March 1971.
2. Jazayeri, M., W. F. Ogden, and W. C. Rounds, "The intrinsic exponential complexity of the circularity problem for attribute grammars," *Comm. ACM* **18**: No. 12, *pp*. 697-706.
3. Reps, T., *Generating Language-Based Environments*, MIT Press, Cambridge, MA, 1984.
4. Horowitz, S. and T. Teitelbaum, "Relations and Attributes: A symbiotic basis for editing environments," *Proc. of SIGPLAN 85 Symp. on Lang. Issues in Prog. Environments*, Seattle, WA, *pp*. 93-106, June 1985.
5. United States Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, Feb. 1983, Springer-Verlag, New York.
6. Wirth, N., *Programming in MODULA-2*, Springer-Verlag, New York, 1983.
7. Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.
8. Giegerich, R., and R. Wilhelm, "Counter-one-pass features in one-pass compilation: a formalization using attribute grammars," *Information Processing Letters* **7**: 6, *pp*. 279-284.
9. Aho, A. V., R. Sethi, and J. D. Ullman, *Complers: Principles, Techniques, and Tools*, Addison-Wesley, New York, 1985.

An Overview of Miranda

David Turner

Computing Laboratory
University of Kent
Canterbury CT2 7NF
ENGLAND

ABSTRACT

Miranda† is an advanced functional programming system which runs under the UNIX operating system. The aim of the Miranda system is to provide a modern functional programming language, embedded in an "industrial quality" programming environment. It is now being used at a growing number of sites for teaching functional programming and as a vehicle for the rapid prototyping of software.

Introduction

The purpose of this short article is to give a brief overview of the main features of Miranda. The topics we shall discuss, in order, are:

- Basic ideas
- The Miranda programming environment
- Guarded equations and block structure
- Pattern matching
- Currying and higher order functions
- ZF expressions
- Lazy evaluation and infinite lists
- Polymorphic strong typing
- User defined types
- Type synonyms
- Abstract data types
- Separate compilation and linking
- Current implementation status

Basic ideas

The Miranda programming language is purely functional — there are no side effects or imperative features of any kind. A program (actually we don't call it a program, we call it a "script") is a collection of equations defining various functions and data structures which we are interested in computing. The order in which the equations are given is not in general significant.

† Miranda is a trademark of Research Software Ltd. An address for Miranda licensing enquiries is given at the end of this article.

There is for example no obligation for the definition of an entity to precede its first use. Here is a very simple example of a Miranda script:

```
z = sq x / sq y
sq n = n * n
x = a + b
y = a - b
a = 10
b = 5
```

Notice the absence of syntactic baggage — Miranda is, by design, rather terse. There are no mandatory type declarations, although (see later) the language is strongly typed. There are no semicolons at the end of definitions — the parsing algorithm makes intelligent use of layout. Note that the notation for function application is simply juxtaposition, as in "sq x". In the definition of the sq function, "n" is a formal parameter — its scope is limited to the equation in which it occurs (whereas the other names introduced above have the whole script for their scope).

The most commonly used data structure is the list, which in Miranda is written with square brackets and commas, eg:

```
week_days = ["Mon","Tue","Wed","Thur","Fri"]
days = week_days ++ ["Sat","Sun"]
```

Lists may be appended by the "++" operator. Other useful operations on lists include infix ":" which conses an element to the front of a list, "#" which takes the length of a list, and infix "!" which does subscripting. So for example 0:[1,2,3] has the value [0,1,2,3], #days is 7, and days!0 is "Mon".

There is also an operator "– –" which does list subtraction.
For example [1,2,3,4,5] – – [2,4] is [1,3,5].

There is a shorthand notation using ".." for lists whose elements form an arithmetic series. Here for example are definitions of the factorial function, and of a number "result" which is the sum of the squares of the odd numbers between 1 and 100 (sum and product are library functions):

```
fac n = product [1..n]
result = sum[1,3..100]
```

The elements of a list must all be of the same type. A sequence of elements of mixed type is called a tuple, and is written using parentheses instead of square brackets. Example

```
employee = ("Jones",True,False,39)
```

Tuples are analogous to records in Pascal (whereas lists are analogous to arrays). Tuples cannot be subscripted — their elements are extracted by pattern matching (see later).

The programming environment

The Miranda system is interactive and runs under UNIX as a self contained subsystem. The basic action is to evaluate expressions, supplied by the user at the terminal, in the environment established by the current script. For example evaluating "z" in the context of the first script given above would produce the result "9".

The Miranda compiler works in conjunction with a screen editor (normally this is "vi" but it can be set to any editor of the users choice) and scripts are automatically recompiled after edits, and any syntax or type errors signalled immediately. The polymorphic type system permits a very high proportion of logical errors to be detected at compile time.

There is quite a large library of standard functions. There is an online reference manual documenting all aspects of the system. There is also a good interface to UNIX, permitting Miranda functions to take data from, and sénd results to, arbitrary UNIX files, and it is also possible to invoke Miranda functions directly from the UNIX shell, and to combine them, via UNIX pipes, with processes written in other languages.

Guarded equations and block structure

In a Miranda script an equation can have several alternative right hand sides distinguished by "guards" (a guard is a boolean expression written following a comma). So for example the greatest common divisor function can be written:

```
gcd a b = gcd (a-b) b, a>b
        = gcd a (b-a), a<b
        = a, a=b
```

The last guard in such a series of alternatives can be written "otherwise" to indicate a default case.

It is also permitted to introduce local definitions on the right hand side of a definition, by means of a "where" clause. Consider for example the following definition of a function for solving quadratic equations (it either fails or returns a list of one or two real roots):

```
quadsolve a b c = error "complex roots", delta<0
          = [-b/(2*a)], delta=0
          = [-b/(2*a) + radix/(2*a),-b/(2*a) - radix/(2*a)], delta>0
                    where
                    delta = b*b - 4*a*c
                    radix = sqrt delta
```

Where clauses may occur nested, to arbitrary depth, allowing Miranda programs to be organised with a nested block structure. Indentation of inner blocks is compulsory, as layout information is used by the parser.

Pattern matching

It is permitted to define a function by giving several separate equations, distinguished by the use of different patterns in the formal parameters. This provides another method of doing case analysis which is often more elegant than the use of guards. We here give some simple examples of pattern matching on natural numbers, lists, and tuples. Here is (another) definition of the factorial function, and a definition of ackerman's function:

```
fac 0 = 1
fac (n+1) = (n+1)*fac n

ack 0 n = n+1
ack (m+1) 0 = ack m 1
ack (m+1) (n+1) = ack m (ack (m+1) n)
```

Here is a (naive) definition of a function for computing the n'th fibonacci number:

```
fib 0 = 0
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

Here are some simple examples of functions defined by pattern matching on lists:

```
sum [] = 0
sum (a : x) = a + sum x

product [] = 0
product (a : x) = a * product x

reverse [] = []
reverse (a : x) = reverse x ++ [a]
```

Accessing the elements of a tuple is also done by pattern matching. For example the selection functions on 2-tuples can be defined thus

```
fst (a,b) = a
snd (a,b) = b
```

As final examples we give the definitions of two Miranda library functions, take and drop, which return the first n members of a list, and the rest of the list without the first n members, respectively

```
take 0 x = []
take (n+1) [] = []
take (n+1) (a : x) = a : take n x

drop 0 x = x
drop (n+1) [] = []
drop (n+1) (a : x) = drop n x
```

Notice that the two functions are defined in such a way that that the following identity always holds — "take n x ++ drop n x = x" — including in the pathological case that the length of x is less than n.

Currying and higher order functions

Miranda is a fully higher order language — functions are first class citizens and can be both passed as parameters and returned as results. Function application is left associative, so when we write "f x y" it is parsed as "(f x) y", meaning that the result of applying f to x is a function, which is then applied to y. The reader may test out his understanding of higher order functions by working out what is the value of "answer" in the following script:

```
answer = twice twice twice suc 0
twice f x = f (f x)
suc x = x + 1
```

Note that in Miranda every function of two or more arguments is actually a higher order function. This is very useful as it permits partial parameterisation. For example "member" is a library function such that "member x a" tests if the list x contains the element a (returning True or False as appropriate). By partially parameterising member we can derive many useful predicates, such as

```
vowel = member ['a','e','i','o','u']
digit = member ['0','1','2','3','4','5','6','7','8','9']
month = member ["Jan","Feb","Mar","Apr","Jun","Jul","Aug","Sep",
          "Oct","Nov","Dec"]
```

As another example of higher order programming consider the function foldr, defined by

```
foldr op k [] = k
foldr op k (a : x) = op a (foldr op k x)
```

All the standard list processing functions can be obtained by partially parameterising foldr. Examples

```
sum = foldr (+) 0
product = foldr (*) 1
reverse = foldr postfix []
        where postfix a x = x ++ [a]
```

ZF expressions

ZF expressions give a concise syntax for a rather general class of iterations over lists. The notation is adapted from Zermelo Frankel set theory (whence the name "ZF"). A simple example of a ZF expression is:

[n*n | n ⟵ [1..100]]

This is a list containing (in order) the squares of all the numbers from 1 to 100. The above expression would be read aloud as "list of all n*n such that n drawn from the list 1 to 100". Note that "n" is a local variable of the above expression. The variable-binding construct to the right of the bar is called a "generator" — the "⟵" sign denotes that the variable introduced on its left ranges over all the elements of the list on its right. The general form of a ZF expression in Miranda is:

```
[ body | qualifiers ]
```

where each qualifier is either a generator, of the form "var ⟵ exp", or else a filter, which is a boolean expression used to restrict the ranges of the variables introduced by the generators. When two or more qualifiers are present they are separated by semicolons. An example of a ZF expression with two generators is given by the following definition of a function for returning a list of all the permutations of a given list,

```
perms [] = [[]]
perms x = [ a : y | a ⟵ x; y ⟵ perms (x—[a]) ]
```

The use of a filter is shown by the following definition of a function which takes a number and returns a list of all its factors,

```
factors n = [ i | i ← [1..n div 2]; n mod i = 0 ]
```

ZF notation often allows remarkable conciseness of expression. We give two examples. Here is a Miranda statement of Hoare's "Quicksort" algorithm, as a method of sorting a list,

```
sort [] = []
sort (a : x) = sort [ b | b ← x; b≤a ]
               ++ [a] ++
               sort [ b | b ← x; b>a ]
```

Here is a Miranda solution to the eight queens problem. We have to place eight queens on chess board so that no queen gives check to any other. Since any solution must have exactly one queen in each column, a suitable representation for a board is a list of integers giving the row number of the queen in each successive column. In the following script the function "queens n" returns all safe ways to place queens on the first n columns. A list of all solutions to the eight queens problem is therefore obtained by printing the value of (queens 8).

```
queens 0 = [[]]
queens (n+1) = [ q : b | b ← queens n; q ← [0..7]; safe q b ]
safe q b = and [ ~ checks q b i | i ← [0..#b-1] ]
checks q b i = q=b!i \/ abs(q - b!i)=i+1
```

Lazy evaluation and infinite lists

Miranda's evaluation mechanism is "lazy", in the sense that no subexpression is evaluated until its value is known to be required. One consequence of this is that is possible to define functions which are non-strict (meaning that they are capable of returning an answer even if one of their arguments is undefined). For example we can define a conditional function as follows,

```
if True x y = x
if False x y = y
```

and then use it in such situations as "if (x=0) 0 (1/x)".

The other main consequence of lazy evaluation is that it makes it possible to write down definitions of infinite data structures. Here are some examples of Miranda definitions of infinite lists (note that there is a modified form of the ".." notation for endless arithmetic progressions)

```
ones = 1 : ones
repeat a = x
           where x = a : x
nats = [0..]
odds = [1,3..]
squares = [ n*n | n ← [0..] ]
perfects = [ n | n ← [1..]; sum(factors n) = n ]
primes = sieve [ 2.. ]
           where
           sieve (p : x) = p : sieve [ n | n ← x; n mod p > 0 ]
```

One interesting application of infinite lists is to act as lookup tables for caching the values of a function. For example our earlier naive definition of "fib" can be improved from exponential to linear complexity by changing the recursion to use a lookup table, thus

```
fib 0 = 1
fib 1 = 1
fib (n+2) = flist!(n+1) + flist!n
           where
           flist = map fib [ 0.. ]
```

Another important use of infinite lists is that they enable us to write functional programs representing networks of communicating processes. Consider for example the hamming numbers problem — we have to print in ascending order all number of the form 2^a*3^b*5^c, for a,b,c≥0. There is a nice solution to this problem in terms of communicating processes, which can be expressed in Miranda as follows

```
hamming = 1 : merge (f 2) (merge (f 3) (f 5))
           where
```

```
        f a = [ n*a | n ← hamming ]
        merge (a : x) (b : y) = a : merge x (b : y), a<b
                = b : merge (a : x) y, a>b
                = a : merge x y, a=b
```

Polymorphic strong typing

Miranda is strongly typed. That is, every expression and every subexpression has a type, which can be deduced at compile time, and any inconsistency in the type structure of a script results in a compile time error message. We here briefly summarise Miranda's notation for its types.

There are three primitive types, called num, bool, and char. The type num comprises integer and floating point numbers (the distinction between integers and floating point numbers is handled at run time — this is not regarded as being a type distinction). There are two values of type bool, called True and False. The type char comprises the ascii character set (em character constants are written in single quotes, using C escape conventions, e.g. 'a', '$', '\n' etc.

If T is type, then [T] is the type of lists whose elements are of type T. For example [[1,2],[2,3],[4,5]] is of type [[num]], that is it is a list of lists of numbers. String constants are of type [char], in fact a string such as "hello" is simply a shorthand way of writing ['h','e','l','l','o'].

If T1 to Tn are types, then (T1,...,Tn) is the type of tuples with objects of these types as components. For example (True,"hello",36) is of type (bool,[char],num).

If T1 and T2 are types, then T1 ⟶ T2 is the type of a function with arguments in T1 and results in T2. For example the function sum is of type [num] ⟶ num. The function quadsolve, given earlier, is of type num ⟶ num ⟶ num ⟶ [num]. Note that "⟶" is right associative.

Miranda scripts can include type declarations. These are written using "::" to mean is of type. Example

```
    sq :: num ⟶ num
    sq n = n * n
```

The type declaration is not necessary, however. The compiler is always able to deduce the type of an identifier from its defining equation. Miranda scripts often contain type declarations as these are useful for documentation (and they provide an extra check, since the typechecker will complain if the declared type is inconsistent with the inferred one).

Types can be polymorphic, in the sense of Milner [Milner 78]. This is indicated by using the symbols * ** *** etc as an alphabet of generic type variables. Example, the identity function, defined in the Miranda library as

```
    id x = x
```

has the following type

```
    id :: * ⟶ *
```

this means that the identity function has many types. Namely all those which can be obtained by substituting an arbitrary type for the generic type variable, eg "num ⟶ num", "bool ⟶ bool", "(* ⟶ **) ⟶ (* ⟶ **)" and so on.

We illustrate the Miranda type system by giving types for some of the functions so far defined in this article

```
    fac :: num ⟶ num
    ack :: num ⟶ num ⟶ num
    sum :: [num] ⟶ num ⟶ num
    month :: [char] ⟶ bool
    reverse :: [*] ⟶ [*]
    fst :: (*,**) ⟶ *
    snd :: (*,**) ⟶ **
    foldr :: (* ⟶ **) ⟶ ** ⟶ [*] ⟶ **
    perms :: [*] ⟶ [[*]]
```

User defined types

The user may introduce new types. This is done by an equation in "::=". For example a type of labelled binary trees (with numeric labels) would be introduced as follows,

```
    tree ::= Nilt | Node num tree tree
```

This introduces three new identifiers — "tree" which is the name of the type, and "Nilt" and "Node" which are the constructors for trees. Nilt is an atomic constructor, while Node takes three arguments, of the types shown. Here is an example of a tree built using these constructors

```
t1 = Node 7 (Node 3 Nilt Nilt) (Node 4 Nilt Nilt)
```

To analyse an object of user defined type, we use pattern matching. For example here is a definition of a function for taking the mirror image of a tree

```
mirror Nilt = Nilt
mirror (Node a x y) = Node a (mirror y) (mirror x)
```

User defined types can be polymorphic — this is shown by introducing one or more generic type variables as parameters of the "::=" equation. For example we can generalise the definition of tree to allow arbitrary labels, thus

```
tree * ::= Nilt | Node * (tree *) (tree *)
```

this introduces a family of tree types, including tree num, tree bool, tree (char $\longrightarrow$ char), etc.

The types introduced by "::=" definitions are called "algebraic types". Algebraic types are a very general idea. They include scalar enumeration types, eg

```
color ::= Red | Orange | Yellow | Green | Blue | Indigo | Violet
```

and also give us a way to do union types, for example

```
bool_or_num ::= Left bool | Right num
```

It is interesting to note that all the basic data types of Miranda could be defined from first principles, using "::=" equations. For example here are type definitions for bool, (natural) numbers and lists,

```
bool ::= True | False
nat ::= Zero | Suc nat
list * ::= Nil | Cons * (list *)
```

Having types such as "num" built in is done for reasons of efficiency — it isn't logically necessary.

It is also possible to associate "laws" with the constructors of an algebraic type, which are applied whenever an object of the type is built. For example we can associate laws with the Node constructor of the tree type above, so that trees are always balanced. We omit discussion of this feature of Miranda here for lack of space — interested readers will find more details in the references [Thompson 86, Turner 85].

Type synonyms

The Miranda programmer can introduce a new name for an already existing type. We use "==" for these definitions, to distinguish them from ordinary value definitions. Examples

```
string == [char]
matrix == [[num]]
```

Type synonyms are entirely transparent to the typechecker — it is best to think of them as macros. It is also possible to introduce synonyms for families of types. This is done by using generic type symbols as formal parameters, as in

```
array * == [[*]]
```

so now eg 'array num' is the same type as 'matrix'.

Abstract data types

In addition to concrete types, introduced by "::=" or "==" equations, Miranda permits the definition of abstract types, whose implementation is "hidden" from the rest of the program.

To show how this works we give the standard example of defining stack as an abstract data type (here based on lists):

```
abstype stack *
with empty :: stack *
        isempty :: stack * → bool
        push :: * → stack * → stack *
        pop :: stack * → stack *
        top :: stack * → *

stack * == [*]
empty = []
isempty x = (x=[])
push a x = (a : x)
pop (a : x) = x
top (a : x) = a
```

We see that the definition of an abstract data type consists of two parts. First a declaration of the form "abstype ... with ...", where the names following the "with" are called the **signature** of the abstract data type. These names are the interface between the abstract data type and the rest of the program. Then a set of equations giving bindings for the names introduced in the abstype declaration. These are called the **implementation equations**.

The type abstraction is enforced by the typechecker. The mechanism works as follows. When typechecking the implementation equations the abstract type and its representation are treated as being the same type. In the whole of the rest of the script the abstract type and its representation are treated as two separate and completely unrelated types. This is somewhat different from the usual mechanism for implementing abstract data types, but has a number of advantages. It is discussed at somewhat greater length in [Turner 85].

Separate compilation and linking

The basic mechanisms for separate compilation and linking are extremely simple. Any Miranda script can contain one or more directives of the form

```
%include "pathname"
```

where "pathname" is the name of another Miranda script file (which might itself contain include directives, and so on recursively — cycles in the include structure are not permitted however). The visibility of names to an including script is controlled by a directive in the included script, of the form

```
%export names
```

it is permitted to export types as well as values. It is not permitted to export a value to a place where its type is unknown, so if you export an object of a locally defined type, the typename must be exported also. Exporting the name of a "::=" type automatically exports all its constructors. If a script does not contain an export directive, then the default is that all the names (and typenames) it defines will be exported (but not those which it acquired by a %include statement).

It is also permitted to write a **parameterised script**, in which certain names and/or typenames are declared as "free". An example is that we might wish to write a package for doing matrix algebra without knowing what the type of the matrix elements are going to be. A header for such a package could look like this:

```
%free   { element :: type
          zero, unit :: element
          mult, add, subtract, divide ::
               element → element → element
        }

%export matmult determinant eigenvalues eigenvectors ...
|| here would follow definitions of matmult, determinant,
|| eigenvalues, etc. in terms of the free identifiers zero,
|| unit, mult, add, subtract, divide
```

In the using script, the corresponding include statement must give a set of bindings for the free variables of the included script. For example here is an instantiation of the matrix package sketched above, with real numbers as the chosen element type:

```
%include "matrix_pack"
                { element == num; zero = 0; unit = 1
                   mult = *; add = +; subtract = -; divide = /
                }
```

The three directives %include %export %free provide the Miranda programmer with a flexible and type secure mechanism for structuring larger pieces of software from libraries of smaller components.

Separate compilation is administered without user intervention. Each file containing a Miranda script is shadowed by an object code file created by the system, and object code files are automatically recreated and relinked if they become out of date with respect to any relevant source. (This behaviour is strongly analogous to that achieved by the UNIX program "make", except that here the user is not required to write a makefile — the necessary dependency information is inferred from the %include directives in the Miranda source.)

Current implementation status

An implementation of Miranda is available for ORION, VAX, SUN, GOULD, Apollo and several other machines running Berkeley UNIX, and also for the AT&T 3B series under system 5. This is an interpretive implementation which works by compiling Miranda scripts to an intermediate code based on combinators. It is currently running at 200 sites (as of January 1988). Licensing information can be obtained from the net address

(ARPA:) "mira-request%ukc@nss.cs.ucl.ac.uk" or
(UUCP:) "mcvax!ukc!mira-request"
(JANET:) "mira-request@ukc.ac.uk" or by real mail from

Research Software Ltd
23 St Augustines Road
Canterbury
Kent CT1 1XP
England

Ports to some other UNIX machines are planned in the near future. Also under study (to appear on a somewhat longer timescale) is the possibility of native code compilers for Miranda on a number of machines, to provide a much faster implementation.

References

Milner, R. "A Theory of Type Polymorphism in Programming" *Journal of Computer and System Sciences*, **17**, *1978*

Thompson, S.J. "Laws in Miranda" *Proceedings 4th ACM International Conference on LISP and Functional Programming*, Boston Mass, August 1986

Turner, D.A. "Miranda: A non-strict functional language with polymorphic types" *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture* Nancy France, September 1985 (Springer Lecture Notes in Computer Science, **201**).

[Note — an earlier draft of this paper first appeared in SIGPLAN Notices, December 1986. This revised version has been prepared for the April 1988 EUUG conference.]

An Overview of the Gothix Distributed System

Alain Kermarrec

IRISA – Campus de Beaulieu –
35042 RENNES-CEDEX - FRANCE –
kermarre@irisa.irisa.fr

1. Introduction

Currently under development at the IRISA/INRIA, GOTHIC [6] is intended to be an integrated distributed system implemented on a network of multi-processor machine BULL SPS7. Since the development of the GOTHIC kernel is assumed to take a rather long time, it was decided to build on UNIX machines (a Network of SUN running under UNIX 4.2 BSD) a system which provides the same interface as GOTHIC in order to start the development of applications. The first release of this system called GOTHIX is currently under test. This paper first describes the concepts developed in both systems and then discusses some implementation details of GOTHIX.

2. The GOTHIX concepts

''Distributed system'' suggests distributed computing and therefore the requirement for specific structures and tools that would allow distributed computation on an object, that is, distributed execution of a single distributed program on several sites. The GOTHIC system provides a new structuring tool for reliable distributed computing: the *multi-functions* [5]. On the other hand it provides objects *fragmentation* and *replication* [1] as a mechanism to distribute the different parts of an object on different sites and to distribute several instances of an object on different sites.

2.1. The concept of multi-function

The concept of *multi-function* is the generalisation of the well-known concept of function or procedure used in conventional progamming language. This is an abstraction (which is a distributed generalisation of the concept of function or procedure) of the notion of block associated with strict rules for communication of parameters and results. A multi-function has its own local variables. A multi-function is made of a certain number of components. Let us consider the following example:

```
multi-function mf (x, y, z : integer), (u, v, w :integer);

var
  <declaration>

cobegin
  (x, y)u: begin ... return u end,      (1)
  (z)v: begin   ... return v end,       (2)
  (y, z)w: begin   ... return w end,    (3)
coend
```

Figure 1. An example of Multi-function

This Multi-function is made out of three components. The first one uses the parameters (x, y) and returns u; the second uses the parameter y and returns v and the third uses (y, z) and returns w.

Two kinds of multi-function calls may be distinguished:

– The *1-p call* where a multi-function is called from a block. Such a call may be pictured as follow:

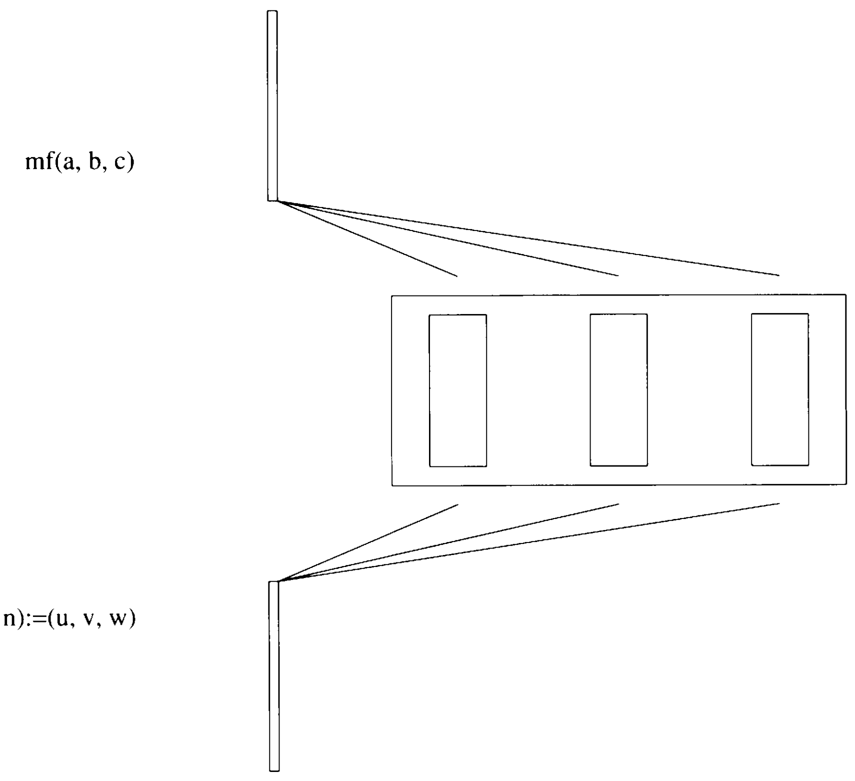


Figure 2. 1-3 multi-function call

The *n-p call* where a multi-function is called from another multi-function. Provided the previously defined multi-function mf let us consider the following program skeleton:

```
cobegin
  (integer a, k, l; ...; (k,l) := mf (x<-a).(u,v); ...)//   (1)
  (integer b, c, m; ...; m := mf(y<-b, z<-c).v; ...)//      (2)
  (integer n; ... ; n := mf().w; ... )                      (3)
coend
```

Figure 3. Nested multi-function call

The execution of the call is shown in the diagram of the figure 4.

In the first case, synchronisation occurs at the end of the multi-function execution when each component returns while in the second case, it occurs both at the call (all the components of the calling multi-function must be ready before the call can occur) and return (all the results must be available before the execution can resume).

The multi-function main advantages are:

- Procedural control structure: from the user point of view there is only one thread of control,
- Parameter/result communication: same as procedure.
- Possibility of concurrent computation.

2.2. Fragmentation and Replication

A distributed system is a set of cooperating computers working together. Distributed computing must take advantage of the distributed system topology. That means that a distributed system must be able to execute distributed programs that operate on distributed data possibly located on different sites. That is the reason while object fragmentation is introduced in the system. An object may be fragmented, that is, made of differents fragments, each of them being located on a different site. At the higher level an object is viewed as an single entity and the fragments are hidden. Once an object is fragmented, each fragment may be processed in a separate way and, therefore, on a particular site and thus concurrently. In order to allow concurrent computation, GOTHIC provides the multi-functions that are, themselves, fragmented objects. Thus, one can develop multi-functions with as many components as the object to be processed has.

On the other hand, an object may be replicated. Several instances of a same object may exist at the same time in the system. In the same way as fragments of fragmented objects cannot be reached at the higher level, users cannot reach a particular instance of a replicated object but only the entire object. It is of the system responsibility to ensure the consistency of the different copies. A complete definition of these properties (illustrated by several examples) can be found in [1].

These concepts are introduced in the POLYGOTH programming language which is the GOTHIC implementation language.

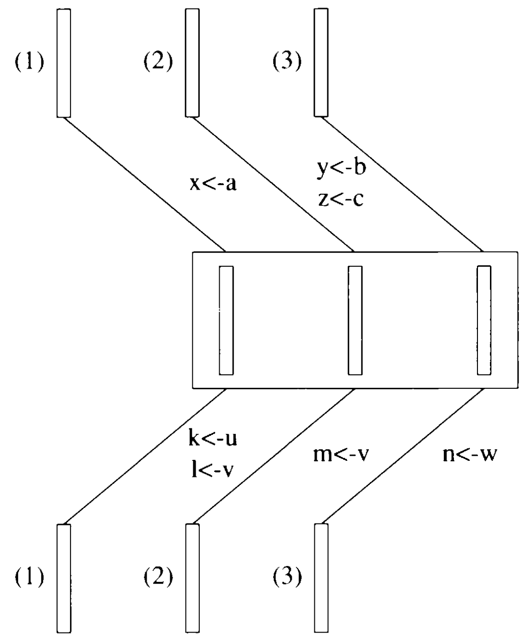


Figure 4. 3-3 multi-function call

3. The Gothix System

The GOTHIX system, built on UNIX, provides facilities to implement objects fragmentation and replication. It is integrated and distributed. From the user point of view, it looks like a traditional centralized system mainframe based system and only the application programs developers will take advantage of the properties offered by the distribution.

3.1. The Gothix object system

Objects provided in GOTHIX are of three types:

1. directories,
2. files or segments,
3. and fragmented files.

All other object are to be built from these kind of objects. For example, a stack will be implemented either ar a file or a fragmented file according whether it is distributed or not. Like in most of traditional centralized system, the object system has a tree structure. This logical structure is the same for the whole system whatever site is considered, that is, the reference to an object does not mention its location. An object is always referenced in the same way. No mount operation [4] is required to access any object of the system. The structure is said to be logical because objects are abstractions of objects not real objects (for example one can read a file, not a specific instance of the file). The organisation of the tree looks like MULTICS [2] or UNIX one.

The nodes of the tree are directories and leaves whatever object of the three types. In general they may be replicated and may migrate. Those that cannot are some system objects that contains information relative to the site (for example the number of the next created file – explained below) The beginning of the tree is the root directory. It is not a super root like in the Newcastle Connection [3]. Each object is reached by its relative or absolute pathname. Absolute pathname are unique through the whole system. The notation is similar to the UNIX notation.

All traditional operations are allowed, according to the access rights.

Replication is a basic operation of GOTHIX because each object may be explicitly replicated and, on the other hand, when a object migrate, it is replicated first on the target site and then deleted on the source site. Further, we have chosen to make objects migrate on the sites they are to be used, that is, each time an object is referenced on a site it is brought to this site and therefore the replication operation is often used. Replication means copy (eventually remote copy) and update of some records (described in a later section).

The operations that are provided on objects are the same as those on traditional systems (*create*, *delete*, *read*, *write* for segments), (*create*, *delete*, *modify*, *look_up*, *change working directory* and so on for directories). Some other facilities deal with access rights.

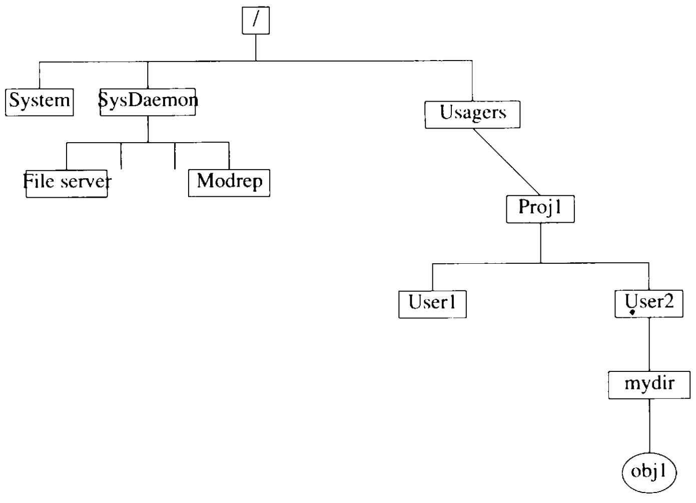


Figure 5. Logical structure of the object system.

3.2. GOTHIX implementation

The GOTHIX system is implemented on a network of SUN machines running under UNIX. Thus we use the facilities provided in the UNIX 4.2 BSD system such as the *sockets* facility, the interprocess communication mechanism on UNIX 4.2 BSD. On the other hand all objects are implemented as UNIX files. The basic characteristics of the implementation are:

- All sites are equivalent. Control is decentralized, no global information is located on a particular site
- All objects are UNIX files and they are all located in the same UNIX directory on a particular site,
- All sites own an instance of the root directory,
- A site owns an instance of the object it currently uses,
- Instances of directories are updated at the same time for each modification that is directory modifications are *atomic*
- Instances of files are updated when necessary that is when a process tries to use an obsolete instance,
- Services are implemented via servers that may communicate.

In this section we describe successively three important aspects of this implementation:

- Object naming,
- Directory structure,
- File access and object consistency.

3.2.1. Objects naming

As mentioned in a previous section, the space name in GOTHIX is represented by a tree structure similar to UNIX one. The beginning of the tree is the root directory. Each object is reached by its relative or absolute pathname. Thus to find an object is the same as finding the UNIX file that matches the GOTHIX object.

On each site, all the objects are located in the same UNIX directory. Each has an unique internal name which is the concatenation of the identifier of the site where it is created and a number. The first created object is given the number zero. This number in incremented each time a new object is created. Objects internal name are never modified (even if objects migrate). It is clear that internal identifiers are unique.

The unique name of the very first created object which is the root directory is the concatenation of the first site identifier and the number 0. Let the site identifier be *vierf*; the unique name is *vierf0*.

Example

Let us consider a system with two sites: *vierf* and *sunlight*. Let the number of the next file to be created be 45 on *vierf* and 32 on *sunlight*. Then if the creation command for the object *toto* is issued on the first machine, the object will get *vierf45* as unique identifier and the number of the next file to be created will be incremented (it gets 46 as new value) since if the command is issued on *sunlight* it will be assigned *sunlight32* and the new value of the number of the next file to be created will become 33.

Note that a number is never assigned twice on same site even if the corresponding file has been deleted.

As there is no site dedicated to specific tasks, every site must be able to locate objects. The information dealing with location is held in directories. Their structure is discussed in the following section.

3.2.2. Directory structure

Each directory provides access to a certain number of objects and hold the informations that concern these objects. A directory is a logical entity made of three UNIX files:

1. The *directory segment* which contains information that cannot be altered.
2. The *status segment* which contains information that can be altered, for example the site vector and the token.
3. The *access segment* which contains sorted access lists of objects located in the directory†.

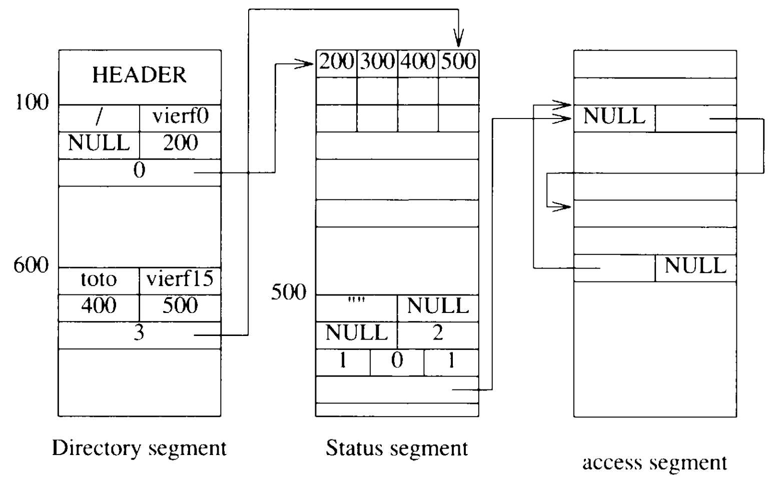


Figure 6. Directory structure

Directory segment structure

The directory segment is divided in two parts:

1. A header that contains informations that deal with the organisation of the segment (first free block, number of free blocks ...)
2. Entries that are C structures that contain:
   - The GOTHIX identifier
   - The name of the corresponding UNIX file
   - The addresses of the next and previous entries
   - The index number of the corresponding record in the status segment.

† Access rights are not considered in this paper and, therefore, we will not describe the structure of this segment. The main difference between GOTHIX and UNIX access rights is that under GOTHIX it is possible to set access to users.

Status segment structure

The status segment is made of an header which is an array of integers. Each item is initialised with the negative value of the address of the corresponding record. When a record is allocated to an object the corresponding array element is given the record address value. Blocks that correspond to positive value of array elements are busy. This is used when allocating a record in the status segment for a new created object in a directory (i.e. when a new entry is created in the directory or a new instance of a object is created on the same site). When the record is freed the array element is given back the negative value of the record.

The second part of the status segment contains records where are registered objects variable informations. They contains the following fields:

- Unix file name. This field is used when several instances of a objects are located on the same site. There is only one corresponding entry in the directory segment and as many records in the status segment as instances on the site.
- Previous and next records number. These fields are used to handle the list of the records that represent the different instances of the same object on the same site.
- Address of the first element of the access list.
- An integer which is the *token* associated with each object,
- The *site vector*,

The *token* is an integer that represents the write permission. When a write operation is issued, the site where this operation occurs must hold the token of the object. The way of getting the token is discussed in a following section.

The *site vector* is an array of bits. Each bit corresponds to a site. A site the corresponding bit of which (i.e the bit index is equal to site number) is set owns at least an instance of the object. If the bit is not set, the site does not own any instance. For example let a site vector be:

site vector = <1, 1, 0, 0, ,1>

if the system has five sites. Then the sites number 0, 1 and 4 own a copy of the object while sites number 2 and 3 do not.

The Figure 6 shows an example of the recording of an object named *toto* (addresses values are not real addresses) according to the diagram of the Figure 7. The system has three sites; there is an instance of *toto* on the site number 0 and on the site number 2 which owns the *token*. The following section presents the way the objects are located and accessed and how the consistency is maintained.

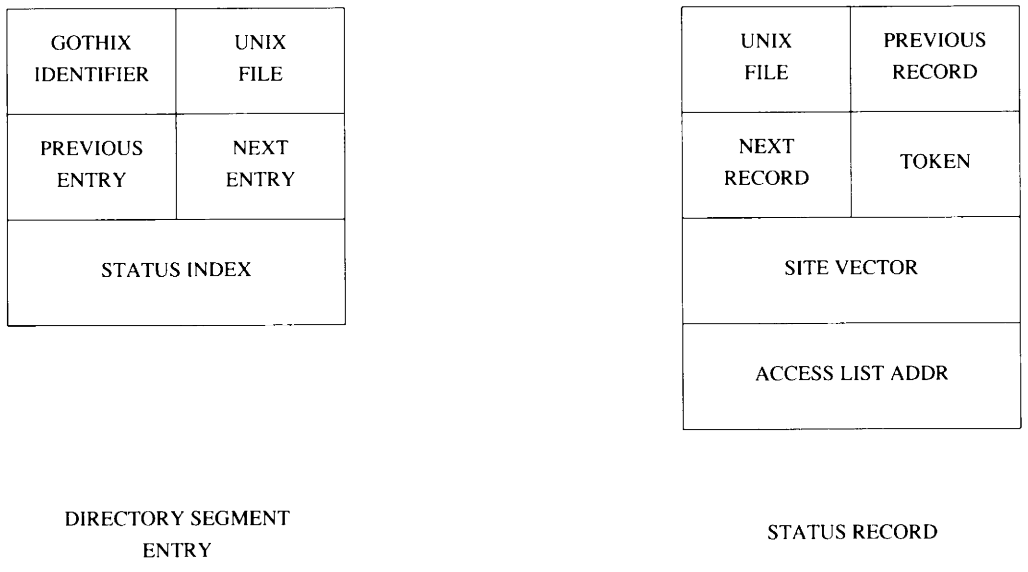


Figure 7. Directory Segment Entry and Status Record

3.2.3. Objects access and consistency

In GOTHIX objects access and consistency are led together. Each site runs the following servers:

- A file server *servfich*
- A lock handler *glock*
- A file transfer server *transfich*
- A token server *servjeton*
- A remote lock handler *glocksd*
- A directory modifier *mdrep*

The location of objects is found in status segment of the father directory which is looked up when a process issues an request for an access to an object. This operation requires an read access to the directory.

Below is briefly discussed the simplified algorithm for objects access.

An access to an object by a process P may be described as follows:

```
{
 P asks the file server the right to perform the operation;
 waits for it;
 if (the requested operation is a read operation)
  {
   performs the operation;
   informs the file server when it is finished;
  }
 else /* Write operation requested */
  {
   if (the write has just to invalidate obsolete instances)
    {
     performs the operation;
     informs the file server when it is finished;
    }
   else /* Atomic update */
    {
     do
       send modification to the file server;
     while (there is still /* at least */ a pending modification);
     issues requests for resetting the locks;
    }
  }
}
```

Figure 8. Object access algorithm

The ***file server*** executes permanently the following loop:

```
{
 Waits for a request;
 creates another process to perform the request;
}

/* Once created the son process performs the following actions */

     {
      locates the object; /* possibly ask the file transfer server to bring a
                                  instance of the object on the site */
      issues a lock /* Exclusive or shared */ request to the lock server;
      Waits for the lock;
      if (request is for reading) /* shared lock */
       {
         gives the right to the process P;
         Waits;
         Receives the end of operation from P;
         issues a request to the lock server to unlock th object;
       }
      else /* write operation requested (Exclusive lock) */
       {
         Issues a request to the token server of the site which is supposed
         to hold it;
         Receives the token;
         Issues a request to the lock server to set an exclusive lock on the
         target object;

         /* for each site that owns a instance of the file */

         Issues a request to the remote lock server of the site;

         if (the write operation just has to invalidate other instances)
          {
           delivers the permission to P; /* Invalidation was carried out by the
                                               remote lock server */
           Waits;
           Receives the end of operation from P;
           issues a request to the lock server to unlock the object;
          }
         else
          {
            /* the operation must occur on all the instances */
           asks P for the operation(s) and transmits them to the modifiers;
           Issues requests to unlock objects;

          }
       }
     }
```

Figure 9. File server

The ***lock server*** algorithm is found in the figure 10. The ***lock server*** handles the locks in a table called the *locked_files_table* (One per site). It does not create any son because it must have the exclusive access to the table.

```
{
 Receives a request;
 if (It is a request for setting a lock)
  {
   if (the concerned file is already locked)
    {
     if (the set lock is exclusive) queues the request;
     else if (the requested lock is exclusive) queues the request;
          else if (number of pending exclusive locks > 0)
                  queues the request;
               else
                  { increments the number of shared locks;
                    informs the requesting process that the lock is set;
                  }
    }
   else
    {
     find a free entry in table of locked files;
     informs the requesting process;
    }

 else  /* request for reseting a lock */
  if (the lock to be reset is a shared lock)
   {
    decrements the number of shared lock;
    if (number of shared lock == 0)
     {
      set the lock for the first pending request ;
       /*
         all the first requests for shared lock that come before the
         first request for a exclusive lock if the first pending request
         is a request for an shared lock
       */
     }
   }
  }
}
```

Figure 10. Lock server loop

The ***remotelock server*** algorithm is shown in Figure 11.

```
{
 Waits for a request;
 creates a son process to handle the incoming request.
}

     /* Son Process */

     {

      Issues a request to the lock server to get an exclusive lock
       on the object;
      if (the operation just has to invalidate the instance)
        {
         invalidates the instance;
         issues a request to reset the lock;
        }

      else /* atomic update */
         informs the requesting process that the lock is set;

     }
```

Figure 11. Remote file server algorithm

Figure 12 shows the ***token server*** algorithm of the token server. Issuing an exclusive lock request on the status segment ensures that the token cannot be given to more than one site.

```
{
 waits for requests;
 creates a son process to handle the incoming request.
}

      /* Son Process */

      {
       issues a request for a exclusive lock on the status segment;
       if (the token is on the site)
        {
         gives back the token to the requesting process;
         updates the status segment;
        }
       else
        gives back the number of the site it has already given the token;

       issues a request to reset the lock;
      }
```

Figure 12. Token server algorithm

4. Current state

The GOTHIX is currently under test on two SUN machines. When it will be considered as reliable, two other SUNs will be added. Adding a new site is not very difficult. It consists in adding a site in a table called *site_table*, creating the GOTHIX directory on the new site and bringing an instance of the GOTHIX root directory on the newly created site. All the following operations are automatically carried out because all objects needed on the new site will be copied as we have explained before. A text editor for fragmented objects has been developed and will be integrated in the system. The design of the POLYGOTH [7] compiler has begun and the development of a document handling application is scheduled for the next six months.

5. References

1. Banatre, J-P., M. Banatre, P. Le Certen, P. Leclerc, F. Ployette. *Fragmentation and Replication in a Parrallel Object Oriented Language.* Rapport do recherche, INRIA. To appear.
2. Organick, Elliot I. *The Multics System: An examination of its structure.* The MIT Press, 1972.
3. Brownbridge, D. R., L. F. Marshall,. B. Randell. The Newcastle Connection or UNIXes of the World Unite! *Software-Practice And Experience.* **12**, pp. 1147-1162, 1982.
4. Walker, B., Gerald Popek, Robert English, Charles Kline and Greg Thiel. The LOCUS Distributed Operating System. *Proceedings the Ninth ACM Symposium on Operating System Principles.* pp. 49-69, 1983.
5. Banatre, J.P., M. Banatre, F. Floyette. The concept of multi-function: a general structuring tool for distributed computing systems. *Proceedings of the 6th DCS, Cambridge.* pp. 478-485, May 1986.
6. Banatre, J.P., M. Banatre, F. Floyette. *An overview of the GOTHIC distributed operating system.* Rapport de recherche, INRIA, March 1986.
7. *POLYGLOTH Le Language de GOTHIC* . INRIA, 1987

A Protocol for the Communication between Objects

R. Schragl
UNA EDV-Beratung GmbH, Muenchen
D. Lauber
Siemens AG, Muenchen

ABSTRACT

In object-oriented systems objects communicate with each other via messages. An object activates processing by sending a message to another object and waiting for its termination. Most of the existing implementations (e.g. SMALLTALK 80) have chosen this procedure. Normally, they are available as stand-alone systems, so that no specific protocols are required. When offering an object-oriented user interface, integrated in a conventional command-oriented system, and with tools running in a local or distributed environment, application protocols are required. This contribution defines a protocol with a service, comparable to the session-layer of the ISO reference model, suitable for this application. The characteristics of the protocol are described, and an implementation is shown within a UNIX system using the programming language C. The concepts are validated in a distributed software development environment, where system software for mainframes is developed using connected workstations based on UNIX.

1. Introduction

Nowadays many systems for PCs and workstations are oriented towards the object-oriented paradigm. This paradigm allows the construction of user-friendly interfaces, usable in environments, where a graphic terminal with a mouse have a considerable influence. One of the best known examples of this family is SMALLTALK 80 ([GOL 80]). Meanwhile, many commercial products (e.g. Apple Macintosh) are available mainly in the area of office automation, as well as in the area of programming languages this paradigm gains more and more interest, as can be seen by the recent developments of e.g. C++, or objective C ([COX 86]).

The fundamental concepts of the object-oriented approach are information hiding, data abstraction, dynamic binding and inheritance ([PAS 86]). Objects are data structures in which the data visible to the user are bound together with the allowed operations (called methods) assuring that only semantic preserving functions are applied to the data. On the other hand, a function is only executed if all data fulfill certain conditions so that complicated semantic data checking is no longer necessary within that function. The set of methods applicable to an object is called a class. Classes are ordered into a hierarchical tree, where nodes on a lower level indicate classes with more specialized semantics.

Objects also have an active aspect besides the abstract data type feature. They act as instances to which messages are sent in order to do a defined piece of work.

In traditional object-oriented systems, the communication is restricted to ''procedure-oriented'' behaviour. This means, that the message containing the name of the method, and the parameters, is sent to the object. The termination of the procedure is indicated to the waiting client. Interactions between these two pieces of information are not possible and there is no supported relation between two procedure calls. This concept is widely used and exists in a lot of systems known as remote procedure call ([TAN 85]).

From a communication point of view, the ISO-Reference-model ([ISO 82]) can be used where the objects are interpreted as level 7 entities. Using a given session service an application protocol can be defined for their communication. For a complete ISO support, the message-based communication has to be mapped onto a session service. The ISO-session-service ([ISO 85]) has performance constraints when supporting short-term sessions (consisting of only a send-request and a confirmation) because it is based on connection-oriented point-to-point sessions with full-duplex bulk data-transfer facilities.

The object-oriented paradigm offers many advantages, so that it should be also used in conventional command-oriented systems. That is why an object-oriented interface should be offered to the user on top of an ordinary operating system to allow the integration of already existing tools running in a local or a distributed environment. An example of this application is the replacing of a ''simple'' terminal connected to a mainframe by a PC or workstation offering a user-friendly unified interface to the complete system.

A tool is defined here as a stand-alone program loaded by commands of the operating system. There are interactive tools with user interface and tools which are acting in the background. The tools can reside in one computer system but it should also be possible that different (also heterogeneous) systems are involved. It should be possible to integrate arbitrary tools in the system without modifying the behaviour of these tools or its user interfaces.

For methods using tools during execution, interactive control possibilities must be provided. This cooperation between objects can be supported by a protocol for object-object communication on which the different application-protocols can be based. In this article such a protocol is described on an abstraction level comparable to the session-layer. The basic service is built in a way that the requirements for object-oriented systems are fulfilled and a mapping to the ISO-session-service is possible.

2. Application model

In this approach the smallest data unit which is interpreted as an object is that piece of information which is conventionally put in a file. This is a restriction to SMALLTALK 80 but it allows tools and users to work with the same data having only different views of them. The change of context between object and file is a task which can be done easily by the system. Further, the restriction mentioned earlier has the advantage of being able to overcome performance constraints for low cost machines because a method can be implemented in a direct way with processes.

Objects are instances of a certain class determining the set of applicable methods. The classes are organized hierarchically with the concept of inheritance to make the implementation of the methods reusable. A method is activated by sending a message to the object. This message contains the name of the method and current parameter values. On receiving such a message the object itself decides which actions have to be taken and which tools (already existing or newly written ones) have to be called.

In order to execute complex methods, interactions between different objects must be supported. A method can initiate the execution of methods of other objects with or without waiting for their completion. The technique allows the definition of new methods by using already existing ones assuring the same kind of semantic checking before a specific tool is executed.

Each object forms a building block with defined connectors. The connectors allow the combination of the objects in different ways. The structure of an object for one method (restricted to its communication behaviour) is given in Figure 1:

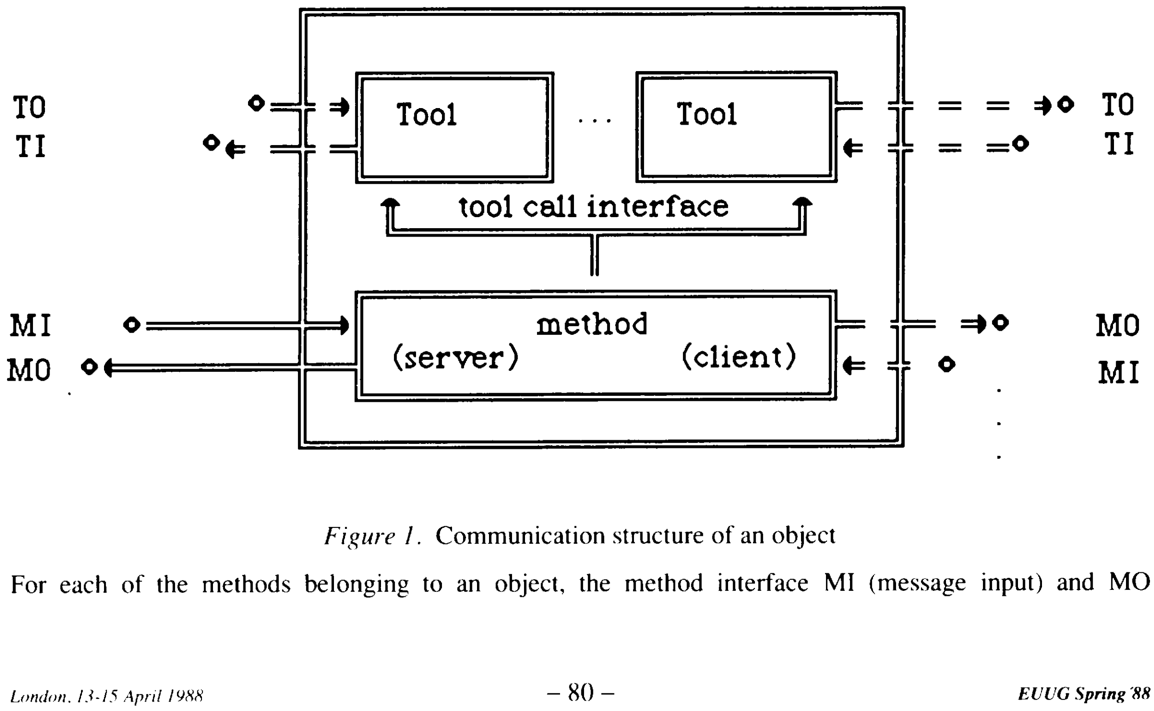


Figure 1. Communication structure of an object

For each of the methods belonging to an object, the method interface MI (message input) and MO (message output) is defined. Via this interface the method acting as server accepts the message for execution. Within a method execution one or more tools can be called, with or without offering an user interface (TI and TO). A method can activate other methods for which it acts as client. For each called method the connectors MO and MI are provided. For efficiency reasons only one MI for all server methods is used, but there exist as many MO as server methods.

When a method is interactively executed all the connectors TO, TI (if available) and MI, MO (of the server part) are bound to the user interface.

When using object-object communication different connections can exist. With two objects, where object 1 is activated by the user, object 2 by object 1, the following possibilities must be supported:

- The tool interfaces of both objects are part of the user interface (Figure 2).

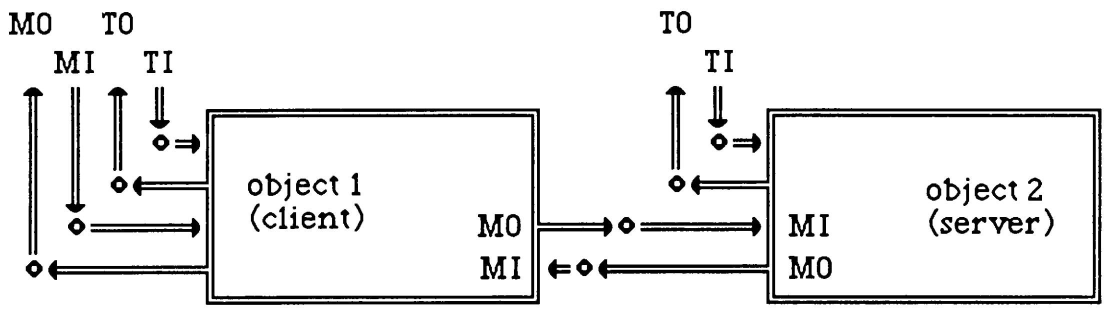


Figure 2. Objects with interactive tools

- The tool interface of the server object (object 2) is used for carrying out a specific application protocol. An example is a file transfer which transfers a file from object 2 to a tool operating within object 1 (Figure 3). It is not necessary that all the connected tools are stand-alone programs, they can also be realized as a function. This allows methods to use defined specific protocols.

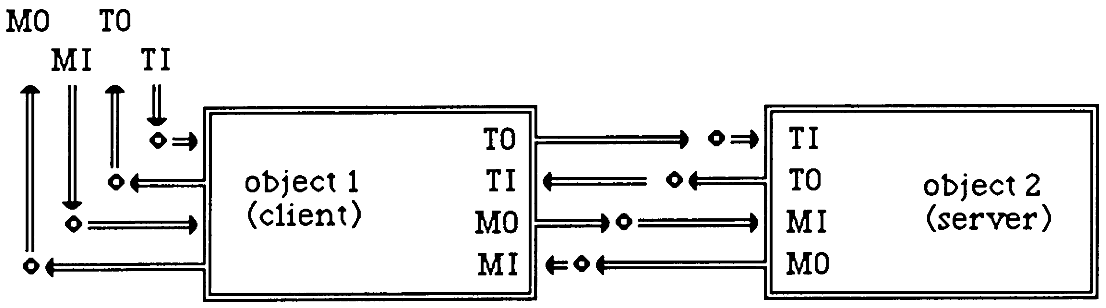


Figure 3. Objects with program interface

- Because the tool interfaces are optional, this connection type must also work with tools without any user interface.
- A client object can activate several methods just as any server can act again as client, so that a hierarchy of executed methods is supported. The individual methods are independent whereby the client decides if the termination of a subordinated server should be waited for or not. This allows a high degree of parallel processing.

The interactions between objects are determined by specific application protocols which have to be defined depending on the tasks performed by the methods. A basic data-exchange service is required which can be used by all objects for the method-dependent communication. This basic service, enabling synchronized interactions between objects, requires the following characteristics:

- A connection is supported between two objects, one acting as a client (the initiator of a method), the other acting as a server (the executor of a method). The connection exists between the beginning of a method and its termination.

- The management of the information about different method activations is carried out by the application, so that this concept corresponds in principle to a ''session'' in sense of ISO. To have a unique term, such a connection is called method-session.
- A control-service for influencing the activities of the server must be supported. This includes primitives for aborting a method, assigning of tokens to the partners (e.g. for suspending and resuming the processing) and sending messages to inform the client about the status of the processing.
- The service must include a name server, which allows an application-oriented identification of the objects. The addressing mechanism must also be able to detect non existing objects.
- Each object must be able to act as a server as well as acting as a client. In both cases, several method-sessions must be supported simultaneously.
- The definition of this protocol presumes a transport service with reliable end-to-end flow control, independency of the communication media (including low speed communication lines), full-duplex connections and expedited data-flow.
- An efficient and conformant implementation should be possible. For PCs and workstations, UNIX plays an important role, so that an implementation on that operating system is selected rather than other operating systems (like MS-DOS or mainframe operating systems).

3. Service and protocol for method-sessions

With the command to execute a specific method, a method-session will be established by the client. It is the responsibility of the server to confirm the start of the execution. One confirmation at most is possible. Within one method-session up to four different data channels will be established, each with its own independent data-flow control and sequencing.

1. The first channel is used for sending control information from the client to the server. The client has the ability to ask for abortion of the server and to resume a suspended method processing.
2. The second channel is used for sending control information from the server to the client. This control information includes the confirmations of the receipt of the establish indication and of method termination (normal or abnormal). The server can suspend its interactive processing by sending a suspend message. This feature enable a method to be sent to the background and suspend the interactive processing. (If a user has a dialogue with a tool at the server side he can interrupt the tool session. That is why the suspended-request is a service primitive of the server).
   Structured messages with user defined data (e.g. for status information) can be sent to the client. Additional message types can be introduced for further development.
3. The third channel is used to send the tool data from the client to the server. All messages are sent transparently without adding protocol information. Therefore, this channel can be used directly by the tools without changing their data flows.
4. The fourth channel has the same characteristics as the third one, but is directed from server to client. In the last two channels (3 and 4) messages can be sent independently of each other.

The sequence of messages is preserved for all channels (there is no need to do so for the first channel). The last two channels are optional and can be chosen dynamically when installing the method-session.

If the transport service fails, an abort-message will be generated and sent to all application entities concerned.

For one method-session, the protocol between one client and one server is given in Figure 4. The protocol behaviour is defined with a petri net, whereby each message is given a transition. The circles indicate the preconditions. Thus the causal structure of the protocol is described without side effects due to time dependencies (cf. [SCH 83]).

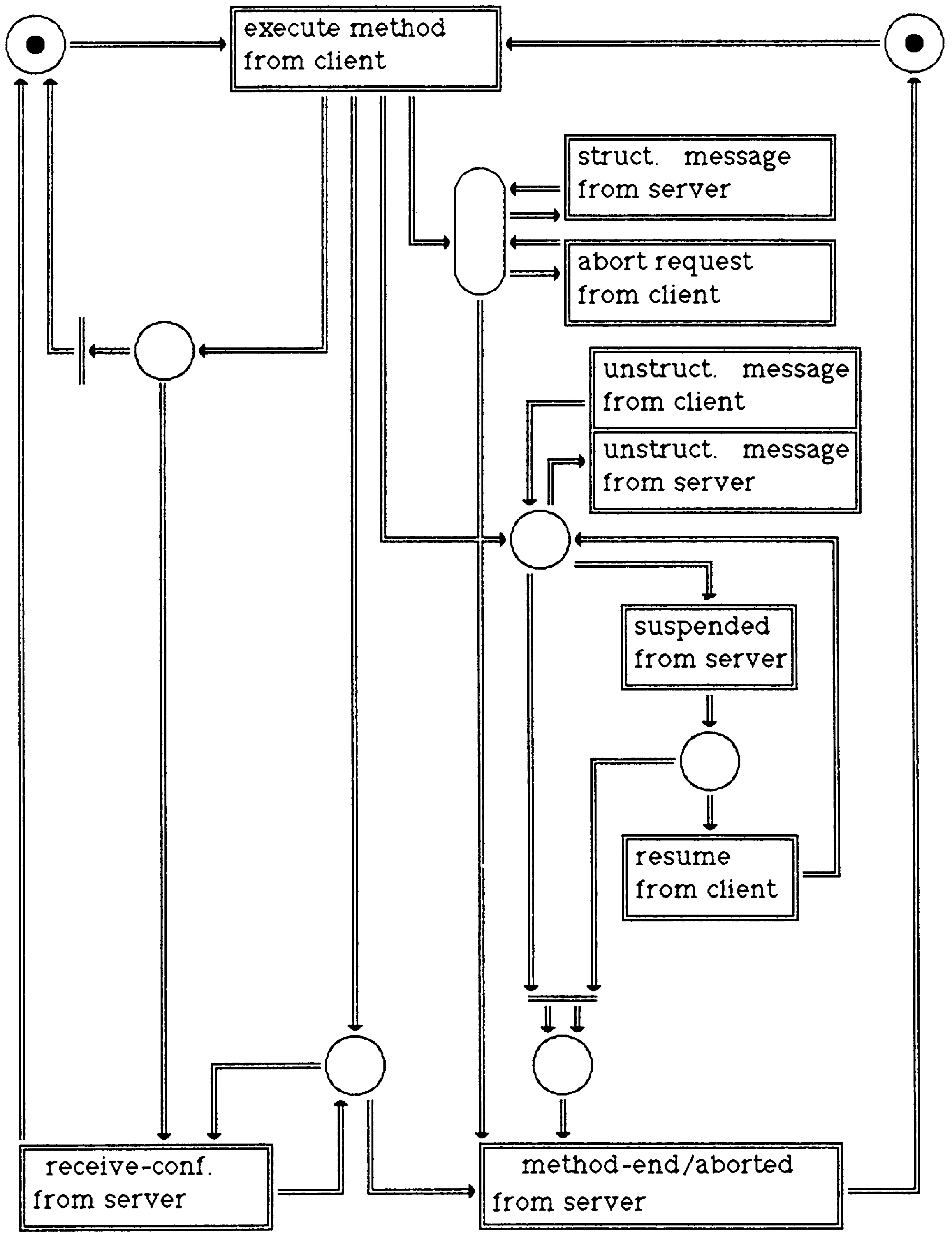

execute method from client
struct. message from server
abort request from client
unstruct. message from client
unstruct. message from server
suspended from server
resume from client
receive-conf. from server
method-end/aborted from server

Figure 4. protocol for object-object communication

The following comments illustrate some peculiarities of the protocol:

- Not more than one confirmation of the establish-request can be given and this is not necessarily the first message of the server.
- The end-message terminates the method-session.
- The server is responsible for the reaction to the abort message.
- The suspend-function can only be initiated by the server.
- After sending a suspend message, no unstructured messages (via channel 3 and 4) are sent until the resume message from the client is received.

The entity which opens a method-session is automatically the client, its partner is the server. An entity can support an arbitrary number of method-sessions. For some of them it acts as client and for others it acts as server. This role cannot be changed within one method-session.

The usual processing at the client's side is the waiting of a response from any of his servers. To avoid (in some systems expensive) scheduling, the response channel (number 2) is built as a special multipoint connection (cf. [SCH 80]). The client controls one endpoint for reading and each associated server controls one endpoint for writing. When the client reads a message it could be sent by any of its servers. To identify the sender of a message, an additional header information in the structured messages is defined. This parameter is set by the client with the establish-request and has to be used by the server when sending a structured message. The service guarantees that all messages from one server are received in sequence.

The format of the most important protocol primitives is given in a Backus Naur similar syntax:

```
<client_msg>      ::=    <execute_method>|<abort_request>|
                         <resume_request>.
<execute_method> ::=    <client_id><object_id><o_id>
                         <method><parameters>.
<abort_request>.
<resume_request>.

<server_msg>      ::=     <o_id><type><m_id><text>.
```

The components of the messages have the following semantics:

```
<client-id> ::=   identification of the client
<object_id> ::=   network-wide unique name of
                  the object
<o_id>      ::=   id of the order (provided
                  by client) and used by the
                  server for struct.  messages
<method>    ::=   name of the method
<parameters>::=   parameter values for a method
<type>      ::=   type of server message:
                  C   confirmation
                  M   user defined message
                  E   method terminated
                  A   method aborted
                  S   method suspended
<m_id>       ::=  id of the message
                  (user defined)
<text>       ::=  content of the message
                  (user defined)
```

4. Implementation concept for using UNIX

UNIX is an operating system with many important concepts like hierarchical file system, multitasking or multiuser facilities. Therefore, this operating system is offered on most workstations. Furthermore, due to the availability of many tools, UNIX is usable as a programmer's workbench as well as a system for developing experimental software. For this reason this operating system has been used to provide a first interface of the service for object-object-communication.

Initially, when implementing a communication service, it has to be decided, which system concepts are meaningful and how they are to be used. In UNIX the following concepts are relevant:

input/output control via stdin and stdout
process (system calls fork and exec)
pipe
signal.

As the method specific processing is based on UNIX commands (arbitrary tools should be used) an individual process is used for each method activation. Methods executed by the same object do not necessarily belong to the same process-family (e.g. processes activated by different users belong to different process-families). If the methods must synchronize themselves, this must be done with object-internal data, if not special UNIX-features (e.g. named pipes) are used. In case of executing interactive tools the method-processes have to belong to the process-family of the user. Therefore, any server is generated as child of a client, so that the common pipe mechanism can be used for the implementation of the channels for the object-object communication.

The object-object-communication primitives are provided as a set of C-functions for the client and another set of C-functions for the server. After the establish-request at the client's side, the function checks if the partner object is available and creates the child process with the program of the server method.

For the information exchange the following UNIX-features are used:

- The establish-request results in a fork (create a new process) and an exec-call (load the program and pass the parameters).
- To submit the control information from the client to the server (channel 1) two signals are used. One to send the abort-request, the other to send the resume-request.
- To send structured messages from the server to the client (channel 3) a pipe is used. At the server's side it always has the file-identification 3, while the client's file-id is determined dynamically. Using the same file-id at different server sites, the multipoint facility is available without additional effort. This simplifies waiting in UNIX for reactions of any of the children.
  With every structured message an additional signal is sent to the client, so that an event driven message processing is possible. The client can ignore the appropriate signal.
- Optionally, the server file-ids for stdin and stdout can be redirected to the client and are used as channels for the unstructured messages (channel 3 and 4). Adding no protocol information to these messages enables the implementation of all functions at user level without enhancement of UNIX kernel functions (e.g. device driver).

If the objects are located in different connected computers a level 4 transport service has to be used. In this case, at least one additional process is required at the server's side managing the transport connection to the network and the pipes to the user process. Depending on the implementation of the transport service, an additional process may also be required at the client's side. Nevertheless, this dependency only influences the internal behaviour of the function and is not visible on the user interface.

If a ISO-session-service is available it can be used by mapping the first two channels onto one session and the last two channels onto an additional session. The second session is only necessary if the connection of the tool-interface has to be supported.

5. Utilization in a distributed software development environment

The object-object communication service was implemented as prototype and validated within SAST ([GEF 87]). SAST is a distributed software development environment offering an object-oriented user interface for using development tools on different operating systems. SAST has been implemented for BS2000 system software developers, to whom the variety of tools on a workstation running under SINIX (which is the Siemens specific variant of UNIX) should be provided without exposing them to two totally different system environments.

An object in SAST is a hidden data structure, mainly resident on a workstation. It is used to describe entities which are relevant in the developer's universe like program sources, documents, tables or configurations. An object consists of a content, describing the user visible data and a frame, representing the applicable methods as a reference to its object class. The object content can consist of different versions of data, including all versions redundantly distributed on a workstation and mainframes ([GEF 86]). Objects can only be addressed by messages. A method refers to a (or a combination of) special tool only in combination with an object. The binding of data and methods make it possible to reduce the number of parameters because of its semantic uniqueness. Besides, objects are able to learn their parameter values.

The communication between user and object is mediated by a dialogue monitor. From the object's point of view this monitor looks like a client to whom the object maintains the same connections as to other client objects. The service primitives allow the monitor to act as administrator of method processes e.g. to manage the change between foreground and background processing, or to control the parallel execution of different methods. The capability for the communication via unstructured messages is not used, so that interactive tools can also be called.

In SAST there are different logical levels for objects. The low level objects are used as windows to externally defined information like a BS2000 as subsystem. The method "dialog" for example calls the terminal emulation. The next level includes objects built on text or source files inheriting basic methods for managing versions and distributing them within a network. The high level objects represent relations to other objects like "to be part of" or "to be associated with".

Depending on the semantic complexity of an object, the methods can use the functionalities of the object-object communication. Within the the prototype the methods for high level objects are able to initiate the execution of methods of the referenced objects by sending messages to them. Examples of this application could be:

- the linear automatic execution of methods at a defined time to achieve better load balancing (e.g. compilation of sources at night).
- tree-structured methods with serial execution to prepare folders where each addressed object (chapter or text) has to return its page number.
- tree-structured methods with parallel execution for software configuration with automatic conditional compilation of sources.

6. Final remarks

For the service of object-object communication an approach was chosen different to the session service of the ISO reference-model:

- Instead of directly using the ISO session synchronization primitives, more simplified mechanisms are used and therefore more data channels are provided. The advantage is that additional application-specific extensions of the protocols can be used beside the defined ones, so that no interference to already existing protocols (and therefore to input/output operations of already existing tools) appear. Nevertheless, it is possible to control the data flow of the additional channels or use application specific primitives.
- An additional feature is the usage of multipoint-connections which help to simplify the implementation. With an adapter-function the multipoint feature can also be mapped onto a point-to-point connection.

The enhancement of the message principle of the traditional object-oriented systems offers some new features:

- Objects with complex semantics require interactive methods which allow the integration of existing tool environments.
- The location of objects is not restricted to only one computer system.
- The usage of existing methods to implement new methods with higher semantics, enables an object-oriented programming. Due to the communication service this is possible in conjunction with a conventional programming technique.
- With help of the service for object-object communication the objects can control the parallel execution of methods also within the object-oriented programming. Due to the constraints of the existing hardware solutions the parallel execution should only be used with strong discipline today. However new hardware technologies using multiprocessors or dedicated processors will support efficient parallel processing which can be utilized in a better way.

The experiences with SAST have proved the protocol's operationality. The object-object communication opens a wide range of applications without too much effort in performance and implementation complexity.

7. Acknowledgements

The authors wish to thank Mr. S. Gefroerer for many helpful discussions and Mrs. Ch. Wimmer who has validated the concepts and implemented the first version of the C-functions.

8. References

[SCH 80]

Schragl, R. *Multipoint Connections in Different Layers of a Communication System* International Conference on Communications, Seattle, IEEE, 1980

[ISO 82]

Basic Reference Model for Open Systems Interconnection, ISO/DIS 7498, 1982

[GOL 83]
Goldberg, A.; Robson, D. *Smalltalk 80: The Language and its Implementation.* Addison-Wesley Publishing Company 1983

[SCH 83]
Schragl, R. *Fundamental Aspects for the Definition of Protocols.* GI-Fachtagung Kommunikation in verteilten Systemen, Berlin, 1983, Informatik-Fachbericht 60

[ISO
Basic Connection Oriented Session Service Definition, ISO/DIS 8326, 1985

[TAN 85]
Tanenbaum, A. S., van Renesse, R. *Distributed Operating Systems.* Computing Surveys, **17**(4), December 1985

[PAS 86]
Pascoe, A. *Elements of Object Oriented Programming.* Byte, August 86, pp. 139-144

[GEF 86]
Gefroerer, E.; Falkenberg, E.; Schragl, R: *SAST - A Distributed Object-Oriented Software Development Environment,* EUUG autumn conference proceedings, Manchester, 1986

[COX 86]
Cox, B. J. *Object-Oriented Programming.* Addison-Wesley Publishing Company 1986

[GEF 87]
Gefroerer, S. *Objektorientierte Entwicklungsumgebung auf SINIX- und BS2000-Basis.* Tagungsunterlagen objektorientierte Sprachen fuer Software-Technologie, CW-CSE, Jan. 1987

Implementation of X.25 PLP in ISO 8802 LAN environments

S.A. Hussain,
J. Ølnes,
T. Grimstad

Norsk Regnesentral,
Blindern
0314 Oslo 3
anwar%vax.nr.uninett@tor.nta.no

ABSTRACT

The X.25 Packet Layer (ISO 8208) and Class II of LLC (ISO 8802/2) are both implemented in the kernel of Berkeley UNIX 4.2BSD on a VAX 11/750 as a new communication domain (AF_XLAN). It is accessible using the IPC primitives provided by 4.2BSD. X.25 PLP's stream services are accessible via stream sockets. Class II of the LLC's datagram services are accessible via raw datagram sockets and stream services via raw stream sockets.

1. Introduction

The standardization work done by the IEEE 802 committee corresponds to the Physical and Data Link layer of the OSI reference model. The need for internetworking LANs with the packet switching data network (PSDN) have resulted in adopting X.25's Packet Layer Protocol (PLP) as the Network Layer (of the OSI reference model) for the LAN protocol stack (*xlan*). Berkeley UNIX, 4.2BSD provides the user with sockets [7] for accessing communication protocols. It also provides structures for creating new protocol domains. Fundamental amongst them is a structure called the protocol switch. This is used in defining the interface between the protocol modules – elements of the protocol stack. What follows is a description, based on [1], of our implementation of the *xlan* protocol stack on the Berkeley UNIX 4.2BSD.

2. User Interface

The user accesses the *xlan* protocol stack thru sockets and the IPC (inter process communication) primitives provided by 4.2BSD UNIX [6]. The domain AF_XLAN provides access to the protocols on this stack. The PLP element [2] provides only virtual circuits. Each end point of the virtual circuit is mapped into a stream socket. To use the PLP, a socket is created using the socket system call:

```
s = socket(AF_XLAN, SOCK_STREAM, 0);
```

The format of the names bound to PLP sockets is derived from the fields of the X.25 Call request packet. It essentially contains the X.121 address of the host machine and a 16 byte user data field. The LLC protocol element provides both connectionless-mode services (Type 1) and connection-mode services (Type 2) as specified in [3]. Access to it is thru raw sockets i.e sockets of type SOCK_RAW[6]. If connectionless service is requested then the raw socket created is datagram oriented. The socket is created by the system call

```
s = socket(AF_XLAN, SOCK_RAW, XL_LLC1);
```

On the other hand, if connection-mode service is requested then the raw socket behaves as a stream socket as it is now an endpoint in a virtual circuit. Accordingly 4.2BSD IPC system calls must be used for setting up a connection before data is sent or received. A raw stream socket to access LLC Type 2 services is created by

```
s = socket(AF_XLAN, SOCK_RAW, XL_LLC2);
```

The format of the names (addresses) bound to raw sockets is given by the *sockaddr_xlan* structure. They are formed by the concatenation of the host's Ethernet address and the LSAP port as shown below.

```
struct  sockaddr_xlan {
        u_short family;
        char    mac_addr[6];
        u_short port;
};
```

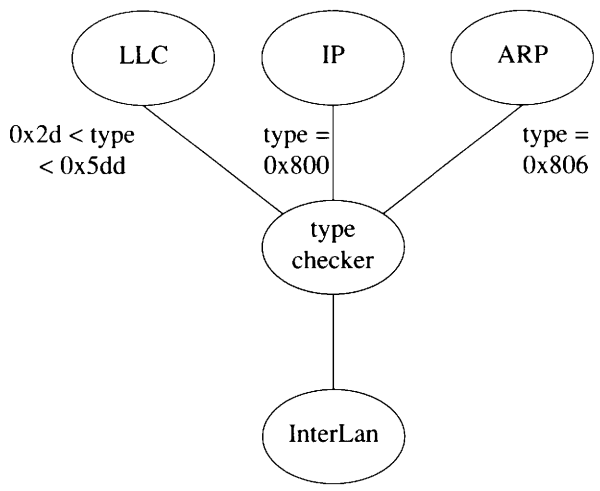
LLC provides 63 LSAP ports which can also be addressed as groups (multicasting).

3. Implementation

In UNIX, an acceptable performance of protocols dictates that it be implemented within the kernel itself. Berkeley UNIX provides several abstractions and utility routines to help create a protocol stack in the kernel [5]. A uniform user interface dictates that one also use the 4.2BSD Socket Manager. The following sections describe the *xlan* protocol stack elements and the interfaces they conform to.

3.1. Medium Access Control (MAC) Level

Our networking environment consists of an Ethernet driven by the InterLan interface. The Ethernet carries DIX (Digital, Intel, Xerox) frames used to implement the IP and ARP suite of protocols. These frames contain a 6 byte source and destination field, a 2 byte type field and the data and CRC fields. The type field indicates the type of data it encapsulates and is used for demultiplexing within the receiving host. IP uses the value 0x800 and ARP uses 0x806 in its type field. The Interlan interrupt routine *ilrint()*, called when a packet is received, uses this field to direct the packet to the appropriate protocol stack. ISO 8802/3 CSMA/CD standard [4], which defines a MAC level protocol, differs from DIX only in its interpretation of the type field. This field now contains the length of the data field. The data field is between 46(0x2c) and 1500(0x5dc) bytes. The fact that this type range is beyond that used by IP and ARP, makes it is possible to differentiate DIX frames from ISO 8802/3 CSMA/CD frames on the Ethernet. This is illustrated in the figure below.



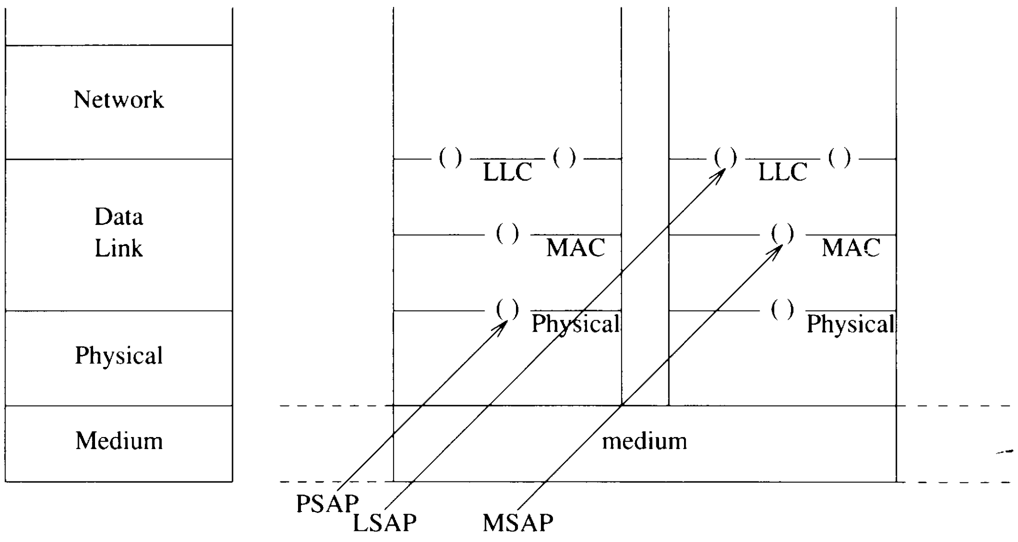
To use the ISO 8802/3 standard as our MAC level protocol, we need only modify the *ilrint()* routine to direct all packets with a type between 46(0x40) and 1500(0x600) to the xlan protocol stack. To do this we define a queue in the kernel, Llcintrq, onto which *ilrint()* will enqueue received ISO 8802/3 frames. The memory buffers or mbuf's provided by 4.2 BSD is used as the mechanism for data storage . The networking code runs off software interrupts. To this end we define a bit in the kernel defined netisr status word, NETISR_XLAN, which is set by *ilrint()* on receiving an ISO 8802/3 frame. An assembly instruction is added to the kernel assembly code to associate the setting of this bit with calling the *llcintr()* routine to process frames on the Llcintrq. Similar structures exist for the IP protocol stack. Thus the setting of NETISR_IP bit in netisr will activate *ipintr()* routine which will process frames on the ipintrq. For outputting an ISO 8802/3 level frame we use Interlan's output routine. 4.2BSD uses struct ifnet to describe all its network interfaces. Each interface has a name. The name of the Interlan's interface is "il0". Handles exist in this structure for the interface's initialization, output, ioctl routines etc.. Thus to call the output routine, one need just map the name to the interface structure pointer and access the output procedure handle. This is shown by the below code.

```
Ifllc = ifunit("il0");              /* mapping routine provided by 4.2 BSD */
(*Ifllc->if_output)(Ifllc, m, dst); /* il_output() */
```

m is a pointer to an mbuf chain containing the data field and dst contains the destination and length field of the ISO 8802/3 frame to be transmitted.

3.2. Logical Link Control (LLC) level

The LLC element of the xlan protocol stack, is the Upper Data link Sublayer of ISO's OSI reference model shown below. Note that while the Network layer (NL) multiplexes NSAPs (Network Service Access Point) onto a single LSAP (Link Service Access point), LLC multiplexes LSAPs onto a single MSAP (MAC Service Access point).



In the context of the OSI model, the services of a layer are the capabilities it offers to the next higher layer. Services are specified by describing the service primitives and parameters which characterize each service. A service is implemented by one or more related primitives which constitute the interface activity. These primitives are of the 3 generic types - Requests(**R**), Indications(**I**) and Confirms(**C**). While **R** is directed by NL to the LLC layer, **I** and **C** are directed by the LLC to the NL. From the standpoint of the NL, R is synchronous while **I** and **C** are asynchronous. The relation between the primitives used to support a service is illustrated by a time sequence diagram.

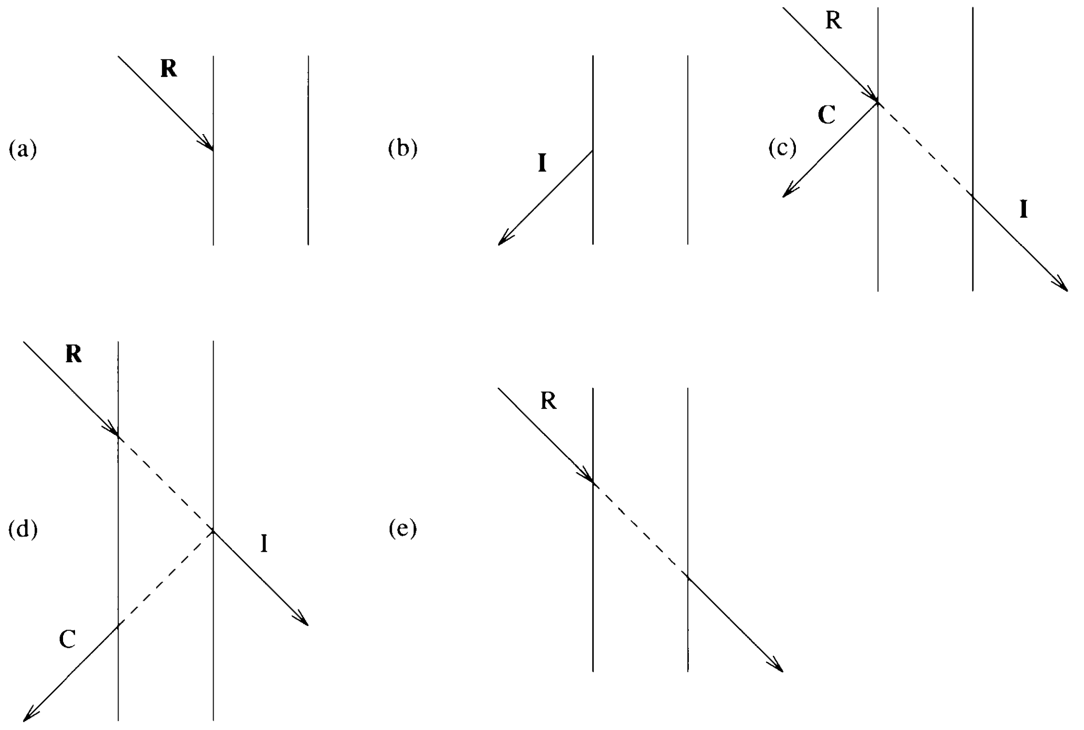
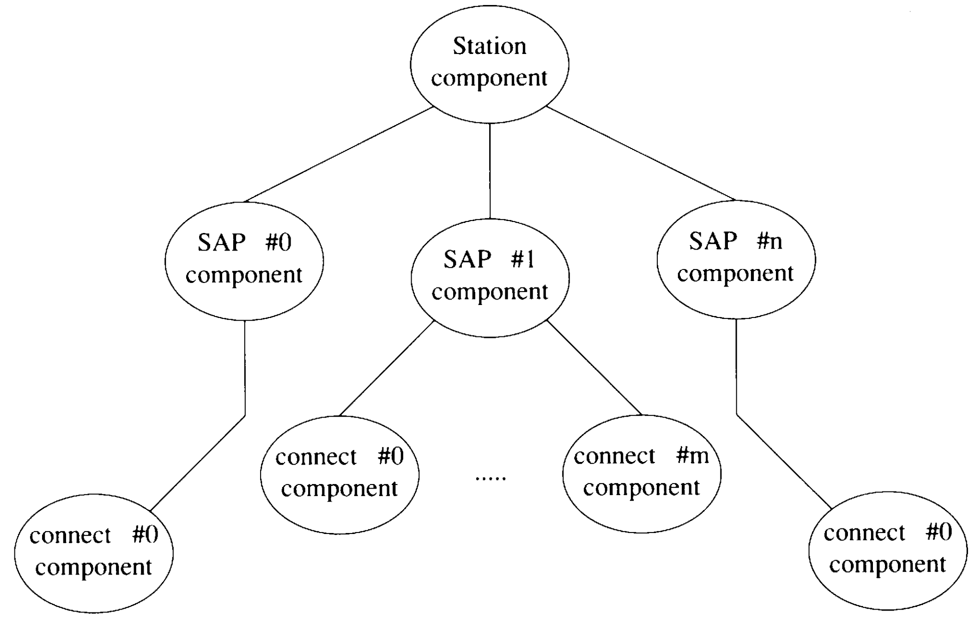


Figure 2. sets of services are provided by the LLC to the NL.

Unacknowledged Connectionless Services which comprises of Point to point, multicast or broadcast Data transfer service(e).
Connection Oriented Services are all point to point. They are: Connection Establishment service(d), Data transfer service(c), Connection reset service(d), Connection termination service(d), Connection flow control service(a) and Report connection status service(b). The references are to the time sequence diagrams above. This scenario is specific to our implementation. To provide these services, the LLC is logically divided into components each characterizing a set of protocol operations defined by a protocol state machine. These components are illustrated below.



1. *Station component*. Essentially this has an UP state and a DOWN state indicating the enabling conditions for the operation of the Service Access Point (SAP) components. This is implemented as a

boolean indicating the existence of the *Interlan* interface.

2. *Service Access Point (SAP) component*. There exist 63 SAP components within the LLC entity. The SAP component has the form:

```
struct  sapcd {
        int             local;          /* SAP address */
        unsigned        grouplist[8];   /* Bit map indicating group */
                                        /* membership */
        struct ccd      *ccd_head;      /* head of ccd list */
};
```

The bits in grouplist are numbered 0 to 255. The bit number is the group number. Consequently a 1 bit in the *nth* position indicates that the SAP is a member of group number *n*. The SAP component dedicates a ccd structure for each data link connection from other LSAPs' to itself. These *ccd*'s form a linked list with the header in *ccd_head*.

3. *Connection component:* This component is described by the *ccd* structure. A data link connection between 2 SAP's is supported by 2 complementary *ccd*'s – one at each SAP. A *ccd* contains a pointer to its LSAP (*sapcd* structure) in *sapcd*. Further all connections terminating at an LSAP are linked together with *ccd_bef* and *ccd_next*. The standard [3] specifies a maximum window size of 128. Local and remote window sizes need not be the same. They are available in *local_modulus* and *remote_modulus* of the *ccd* structure. This implementation sets the local window size to 7 and defaults the remote window size to 7. The latter is negotiable by exchanging XID frames. An abbreviated form of the *ccd* structure is given below.

```
struct  ccd {
        struct ccd          *c_next, *c_bef;      /* for linking ccd specific to a LSAP */
        struct sapcd        *sapcd;               /* back pointer to parent */
        struct timer_node   *timerpool;           /* address of an array indexed by */
                                                  /* timer type */
        struct frmr_data    *frmr_data;           /* buffer for assembling the FRMR */
                                                  /* info field */
        struct mbuf         *retxpool[8];         /* for data retransmission. Each */
                                                  /* mbuf contains 16 pointers to mbuf */
                                                  /* data chains */
        char                local_modulus;        /* local window size + 1 */
        char                remote_modulus;       /* remote window size + 1 */
        char                link_state;           /* state of the finite state machine */
        char                vs;                   /* V(S) */
        char                vr;                   /* V(R) */
        char                last_recvd_nr;        /* lower window edge of local end */
        char                last_transmitted_nr;  /* lower window edge of remote end */
                                                  /* size 128 maintained by retxpool */
        char                lbusy;                /* local LSAP busy flag */
        char                rbusy;                /* remote LSAP busy flag */
        ...
};
```

The *ccd* has a *timer_node* for each timer type (Ack timer, P timer etc.). These are maintained as an array of *timer_node* structures in the *m_dat* field of an *mbuf*. A pointer to this array is available in *timerpool*. Starting a timer puts the corresponding *timer_node* onto a chain which is manipulated by timeout routines. The fields *vs*, *vr*, last *recvd_nr* and last *transmitted_nr* maintain the sliding window. Data is buffered until its acknowledgement is received or connection is reset or disconnected. Data is buffered by storing its *mbuf* pointer in retxpool. retxpool is a logical table of size 128. It is maintained by 2 indices – the primary index points to the *m_dat* field of an *mbuf*. The *m_dat* field has space for 16 data pointers. The number of *mbuf*'s allocated to hold data pointers (max 8) is (*local_modulus*/16). In 4.2BSD, protocol modules use the "protocol switch" to interface with each other. The protocol switch is defined by *struct protosw* in [5]. Fields in this structure define the type of sockets the protocol module supports, the protocol family and handles to the module's routines. Each LSAP, besides LSAP 0 [3], can be used to support a protocol module or can interface directly to the socket manager providing raw sockets. Based on the destination LSAP in the received packet the appropriate protocol switch is used. Presently LSAP #63 is reserved for the PLP. Remaining LSAPs are available to the user for raw sockets. Network layer protocols make the connect, disconnect, reset service request to thè LLC by using their *pr_ctloutput* handles. Indications and confirms are channelled to the Network layer by the LLC by using its *pr_ctlinput* handle.

The below calls show how protocol switches are used by level 3 to request services of the LLC.

```
(void)(*protosw[].pr_output)(0, m, src, dst);   /* Connectionless Data transfer service */
struct     mbuf *m;
struct     sockaddr *src, *dst;
```

m is the *mbuf* chain containing the data.

```
ccd = (*protosw[].pr_ctloutput)(req, src, dst, 0);   /* if req = X_CONN then this */
int  req;                           /* is a Connection Establishment service */
struct     ccd *ccd;
struct     sockaddr *src, *dst;

err = (*protosw[].pr_output)(ccd, m);      /* Connection oriented data transfer service */
struct     ccd *ccd;
struct     mbuf *m;
```

ccd refers to the link connection, and *m* is the *mbuf* chain containing the data. Also, this conforms with the time sequence diagram c. The LLC passes data up towards the user by using the *pr_input* handle of the protocol switch belonging to the destination LSAP. We use the convention

```
(void)(*protosw[].pr_input)(0, m, src, dst);    /* Connectionless data transfer service */
struct     mbuf *m;                 /* This is an Indication */
struct     sockaddr *src, *dst;
```

m is the mbuf chain containing the data.

```
(void)(*protosw[].pr_input)(ccd, m); /* Connection oriented data transfer service */
struct     ccd *ccd;
struct     mbuf *m;
```

ccd refers to the link connection, and *m* is the *mbuf* chain containing the data. Indications and Confirms travel up towards the user. Calls to this purpose are

```
(void) (*protosw[].pr_ctlinput)(ccd, primitive, ic, arg);
struct     ccd *ccd;
int  primitive, ic, arg;
```

where primitive can take values X_CONN, X_RESET, X_STATUS, X_DISCONNECT and ic can take values X_INDICATION, X_CONFIRM. arg contains the status specific to primitive and ic. Besides the protocol switch, Berkeley UNIX defines additional structures for using the raw socket interface. Every socket has a data structure given by *struct socket* [5]. In addition, every raw socket has a protocol control block given by *struct* rawcb [5]. A pointer to this *rawcb* is stored in the field *so pcb* of the socket structure. The field *rcb_pcb* in *rawcb* contains a pointer to the corresponding *ccd* for connection oriented raw sockets and contains a 0 for connectionless sockets. All the rawcb's are kept on a doubly linked list by the kernel and used for packet dispatch. The LLC provides counterparts to the routines *raw_usrreq*, *raw_ctlinput*, *raw_input*, provided by 4.2BSD, to support connection oriented raw sockets. These routines are entered in the protocol switch for LSAPs supporting raw sockets.

3.3. Network level

X.25 is a standard drafted in 1976 by the CCITT for standardizing the interface between a packet-switching data network (PSDN) and the user's data terminal equipment, or DTE. It consists of three protocol levels – the physical level, LAPB, and the Packet Level protocol (PLP). It describes the interface almost entirely from the network's viewpoint and leaves open many issues the DTE must take into account. These issues occurring at the packet layer include DTE timers, cause and diagnostic codes DTE state transitions and DTE-DTE communication without an intervening PSDN. In 1980, a standard, ISO 8208 was introduced to resolve these issues. With the prevalence of both Local Area Networks (LANs) and X.25 equipment, the need for internetworking prompted the adoption of X.25 PLP as the protocol for ISO's Network Layer. The standard ISO 8881 [2], a specialized version of ISO 8208 was introduced to address the special needs of the packet layer in a LAN. X.25 PLP provides the OSI Connection-mode Network Service (CONS) in a LAN station. Each LAN station acts as a DTE. The PLP operates one Packet Level Entity for each DTE/DTE interface in which it is involved, i.e., for each station with which it communicates. The Packet Level entity is identified by the pair of MAC addresses of the two LAN stations associated with the interface.

The Packet Level Entity described by *struct plp_entity* (abbreviated form given below) are all chained together

```
struct  plp_entity {
        struct          plp_entity *next_entity;    /* list of entities on this LAN station */
        char            state;                      /* states of the finite state machine */
        bool            mode;                       /* Is this entity a DTE or DCE */
        struct          lcd *channels;              /* head for list of logical channel */
                                                    /* descriptors */
        struct          sockaddr_xlan remote;       /* identifies the PLP entity */
        struct          ccd *ccd;                   /* the LLC's ccd used by the entity */
        bool            fast_select;                /* per entity facility */
        ...
};
```

using the next_entity field. If the entity is using connection-mode services of the LLC the ccd field contains the corresponding ccd pointer. If connectionless-mode services of the LLC are used the ccd field is zero. Other fields of this structure define the per entity X.25 facilities used by the entity, for example, extended packet sequencing, incoming/outgoing calls barred, etc.. PLP entities multiplex virtual circuits, each described by the *struct lcd* (logical channel descriptor) below.

```
struct  lcd {
        struct  lcd                     *next_lcd;
                                        /* chained together on this field */
        struct  plp_entity *entity;     /* back pointer to PLP entity */
        struct  socket                  *so;
        struct  x25_addr *local, *remote;
        int     channel;                /* logical channel number */
        int     state;
        struct  timer_node *timerpool;  /* array of X.25 timers */
        int     packetsize;             /* max packet size of incoming packet */
        ...
};
```

4.2BSD sockets are the end points of the virtual circuits. Each *lcd* contains a pointer to the socket it supports in so. The *lcd* structure contain fields for maintaining the sliding window and fields for supporting the X.25 per channel facilities, for example, flow control negotiation, fast select, reference numbers, etc. At connection time the PLP tries to use LLC connection-mode procedures. If this fails it uses connectionless-mode procedures. The mapping of an X.121 address to a MAC address is done by using a static table. The use of Reference numbers allows a more elegant algorithm to be used. In this case if the remote's MAC address is unknown, a CALL is sent in broadcast mode. All LAN stations will receive the packet and the one having a user listening for the X.121 address will respond a CALL ACCEPTED carrying the desired MAC address.

4. Acknowledgements

This work was sponsored by The Norwegian Telecommunications Administration and Research Establishment (NTARE).

5. References

1. T.Grimstad, J.Ølnes, S.A.Hussain. *Implementation of X.25 PLP in ISO 8802 LAN environments.* TR 801, Norwegian Computing Center. 1987
2. ISO/8881 – Use of X.25 Packet Layer protocol (ISO 8208) in ISO 8802 LAN. 1985
3. ISO/DIS 8802/2 – IEEE Std. 802.2 Logical Link Control. 1985
4. ISO/DIS8802/3 – IEEE Std. 802.3 CSMA/CD. 1985
5. S. J. Leffler, et. al, 4.2BSD Networking implementation Notes. 1983
6. S. J. Leffler, et. al., 4.2BSD Interprocess Communication Primer. 1986
7. UNIX Programmer's Manual 4.2BSD, Virtual VAX Version. 1983

UNO: USENET News on Optical Disk

A. Garibbo,
L. Regoli,
G. Succi

University of Genoa
Italy

1. Introduction

The size of a WORM optical disk is greater than a Gbyte and it is likely to grow fast within few years; moreover storing and retrieving USENET news is becoming tedious and difficult: at present time a user has easy access to news if he knows exactly which ones he wants to consult; besides reading daily news takes little time using standard read-news tools.

Troubles arise when one wants to find some news only knowing few features because the help he has is merely a hierarchical organization of the news supplied by the USENET system: actually, such a tree-shaped framework seems to be quite unsuitable as long as :

i) the structure is not strongly enforced

ii) quite different leaves lie in the same directory

Owing to the high rate of news traffic, lots of space is needed, and usually each local network connected with USENET either devotes too much space to archiving or it needs frequent backup on tape.

UNO – USENET News on Optical disk – attempts to solve this kind of problems, since more than four years of full news, at the present rate, can be archived on a WORM disk.

All facilities provided by standard readnews tools are enclosed in UNO; moreover it supports an incremental knowledge driven search, which allows interactive data retrieving without either knowing exactly the wanted news or having to deal with all the news of a USENET directory.

UNO provides easy interaction through a smart query language, remote query and intelligent programmable selection of relevant news.

UNO was developed an a workstation named Arianna, based on a National 32032 processor, which runs UNIX System V.3. A WORM disk is fully integrated in the global file system; UNO is designed in C++ according to object oriented programming and software engineering criteria.

2. Object Abstraction in UNO

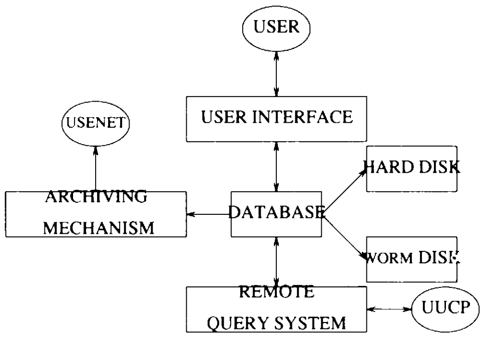
As written above, UNO is designed in object oriented programming style, which means:

2.1. Fully modularity

Four are the big independent modules that build up the skeleton of UNO:

- *database*
- *remote query system*
- *user interface*
- *archiving mechanism*

They interact with one another via message passing, e.g. calling member functions and storing data on files. In the pictures below the data flow through the system is shown

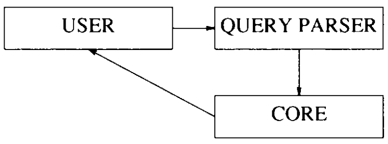


As can easy understood such a modularity allows substitutions and improvements in each module without having to be concerned with the global system consistency.

2.1.1. Data abstraction

Data abstraction implies easy definition of modules global architecture and clean code where errors can be easy detected, the use of C++ programming language, in a such a context, sounds as a natural choice: C++ indeed supports single and static inheritance rules often useful, e.g. in ordering access lists.

3. Database Structure



Two different levels are identified in our database, as it is shown in the picture above

- Query parser
- Core

3.1. Query parser

Its aim is to provide the user a simple interface with database: its input is an understandable request while its output is a query easy to be processed by the database core.

In other words, the query parser translates the query into a core input format through an interaction with the user kernel interface.

The structure of the core query is really simple, as can be argued by its class definition and by the following example:

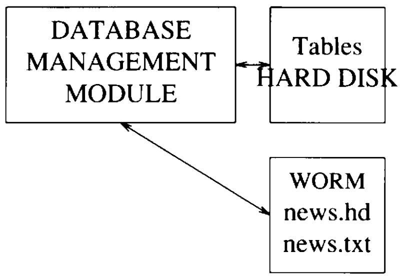
```
class query {
        char *arg[max-key];
        :
        :
public:
        :
        void set_subject_id(char *c);
        :
        void set_author (char *c);
        :
        void set_keyword (char *c);
        :
        istream& operator >> (istream& s);
        ostream& operator >> (ostream& s);
};
```

Example: mapping of request of news with author Ritchie and subject "pic"

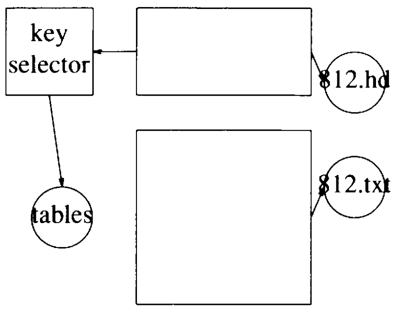
Query.arg	
date	-
country	-
author	Ritchie
subject	pic

Once the core query is compiled, a pattern matching with the news header is fired.

3.2. Core structure



News headers and news are stored in different files to supply a more flexible structure. Several tables – one for each key-field – are used to build a virtual associative memory with which the query parser interacts.



Because of their size, those tables are dynamically loaded in RAM only when it is needed, during query processing.

Tables data structure follows with an example.

```
class couple {
        friend class table;
        char *c;
        header ptr;
        couple *next, *previous;
        :
};

class table {
        :
        couple *head, *index;
        virtual list *read_table (char *expression);
};
```

Example: table for "NAME" field

NAME	FILE
Kernigan	123428.hd
Sinatra	8875.hd
Ritchie	2012.hd
-	-

Tables are sorted in increasing order of specificity, for example, the field "author" is more specific than the field "country" (several authors come from the same country!).

The query processing mechanism acts as follows:

- Table for the most specific field set by the user is loaded in RAM.
- A first pattern matching is performed on this table to create a list of news including the required ones.
- The previously loaded table is removed from RAM.
- Pattern matching, on the other fields specified by the user, in decreasing order of specificity, is performed on news header files pointed to by **header** field of **class couple** instances belonging to the previously made list.

The relational database is designed with a smart splitting of data between hard and optical disk. Headers and news are saved on the optical disk, while tables and several other suitâble information are stored in the hard disk both because such data need to be update (a Write Once Read Many disk is used!) and because of the frequent access to them (the access time to the WORM disk is about five times greater than to the hard disk).

When the optical disk is nearly full and it has to be substituted, the tables are automatically saved on it, whereas at installation of an already used WORM disk (e.g. for consultation) the database in the hard disk, copies them from the optical one, if they exist.

4. User Kernel Interface

Our aim was to make the search through the database nice and so provided a series of masks the user has to deal with according to one of two possibles custom files, which can be enclosed in the home directory of the user: **.uno_db** and **.uno_su** ; the file called .uno_db must be written following the guidelines of a template provided by the designers, it has a higher priority than the other .uno_su which can be defined at runtime using one of the options of the system main menu.

Example 1: Structure of .uno_db if user wants to select

- author field
- dept field

but not one country one.

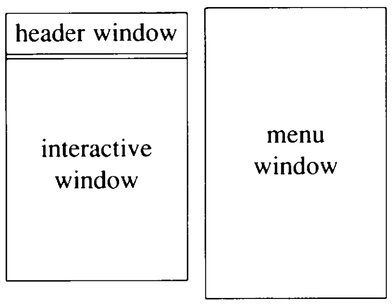
```
# if you want to select author, write it below
        author
# if you want to select country, write it below

# if you want to select dept, write it below
        dept
        :
        :
```

Example 2: Structure of .uno_su (the situation is the same as the example 1)

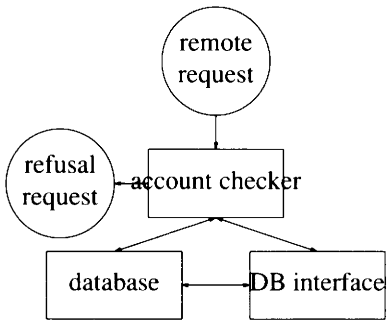
```
author
dept
:
```

Be careful of the fact that this file acts as internal form, so the user is never request to edit it: UNO allows the user to see and to update it runtime.

The screen is structured in windows as can be argued from the following picture



5. Remote Query System



The remote query system consist of two parts: the first one is the account checker that checks the validity of the user request, updates accounting tables and sends warning messages to behindhand users; the second one is the database interface that processes the requests and sends the answers.

The account checker policy is the following:

- Each user has an initial amount of resources which decrease whenever he requires a transaction, proportionately to the size of the interchanged information packed.
- A credit – up to 5% of the starting quota is granted.
- Warning messages are sent when the resources amount becomes less than 10%, 5%, 2% of the starting one. Moreover a warning message is sent for each request as long as credit is not exhausted.
- All the requests are refused when the credit is exhausted.

Database interface works in three steps: first it translates the requests from the user format in a one suitable for the database, then sends a message to the account checker and it creates a query instance that it submits to the database and finally returns data files to the remote user.

Remote requests can be performed using a supplied mail tool, similar to the user kernel interface, but written in C language instead of C++ to make it fully portable, such a tool can also mail a query, via UNIX system call, to a receiver, whose address is specified in one of the custom files; in our implementation the receiver is

The user can write the query by himself, anyway, taking care of the fact that badly formed requests are rejected.

Example 1: well formed request

```
       :
author : Ritchie
   subject : pic
       :
```

Example 2: badly formed request

```
Dear Mr. UNO,
      would you like to send me all
    Ritchie's news about "pic"?

      Yours faithfully,

          LOUIS GHIROUS
```

6. Archiving mechanism

Archiving of news is performed by a self activating time driven daemon which works once a day.

News are stored in files by the mail system, so the daemon recognizes them by reading the date field of the file descriptor. If the date is greater than the date of the last activation of the daemon – previously stored in a global variable – the system processes the news. In this case a pattern matching is performed between news and system custom file, that contains directives about interesting keys.

If the news results interesting, they are archived into the database, according to the mechanism described in section 3.

The custom file,which contains selection directives, can be updated runtime by the superuser without editing it by mean of a specific command provided by UNO package.

7. Conclusion

We think UNO is quite remarkable both because of the idea of building a big and user-friendly database where news that otherwise could be lost can be saved and because it can be further improved e.g. using sockets to perform remote query in a local network and applying A.I. techniques in the selection task.

Unfortunately it is more than an year that no news arrives in Italy, so at present time our system cannot offer the prehaps most interesting news.

8. Acknowledgements

We wish to thank Prof. Joy Marino (University of Genoa - DIST) because he gave us the idea of this project, encouraged us and helped us whenever we needed it.

We also wish to thank professors Antonio Boccalatte and Giorgio Olimpo of University of Genoa.

9. References

1. Aho, A.V., R. Sethi, J.D. Ullman *Compilers: Principles, Techniques and Tools,* Addison Wesley 1986
3. Bishop M. *''How to Use USENET Effectively''. 1986*
4. Cox, B.J. *Object Oriented Programming. An Evolutional Approach* Addison Wesley
5. Glickman, M. *''USENET Version B Installation''*. 1986
6. Horton M.R. *''Standard for Interchange of USENET Messages''*. 1986
7. Stroustroup, B. *''C++ Programming Language''*. Addison Wesley 1986

behindhand file,which

Design of and experience with a software documentation tool

José A. Mañas
Tomás de Miguel
Dept. Ingeniería Telemática
E.T.S.I. Telecomunicación
Ciudad Universitaria
E-28040 MADRID
SPAIN

jmanas@goya.uucp
tmiguel@goya.uucp

ABSTRACT

A UNIX tool is presented that permits to write documented code in a text oriented fashion, looking for humans that have to read, understand, and maintain it, rather than thinking for language processors that have to compile it. The tool permits handling any text processing system, as well as any target language. Several files may be documented and maintained as a single unit, thus helping in keeping them coherent. The tool may easily and productively interact with standard UNIX tools. The design criteria, basic features, and some real examples of utilization are presented.

· · · surely nobody wants to admit writing an *illiterate* program. D.E. Knuth, Literate Programming.

1. Introduction

Software documentation is quite a big problem in educational and research institutions as universities. A strong temptation to laziness on documenting creeps around software that is not to be seriously maintained, nor has to pass severe acceptance tests. Laboratory assistants waste plenty of time decrypting programs written by novice students with very peculiar structuring ideas. Plenty of good pieces of software are lost due to lack of good documentation. On the other hand, students get bored with standard text editors after moving bunches of code around, restructuring the program one thousand times, and so on. In actual practice, resource availability is always below the desires, and there is a finite time for editing before deadlines. In the end, there is a badly presented stuff.

While well documented pieces of software may undertake maintenance, undocumented ones can just be thrown away and rewritten.

All these problems may be overtaken with proper methodology and tool support. Proper tools favour powerful methodologies. While text editors are useful, something more is required for proper documentation. A documentation methodology is required that provides easy and flexible input, as well as fast and beautiful output. Tools promote the use of a methodology for high quality programming.

D.E. Knuth [Knuth, 84] proposed a language, WEB, that mixes $\TeX$ [Knuth, 84a] and Pascal, as documentation and target languages respectively. Following his pioneering work, a documentation methodology has been devised, a more flexible tool has been developed to support it. It permits mixing any number and class of languages, either text processing or programming ones, as well as the tool language itself. Among its features there should be highlighted its ability to cope with many (coherent) output files, smooth interfacing with *make*, easy coupling with standard macroprocessors, linking with language processors and debuggers, and so on. The tool may be easily parameterized to fit peculiar user's tastes or just the text processing language conventions. The tool becomes a valuable filter that may be fruitfully combined with many other UNIX tools.

The paper shall present the design criteria of the tool, features, several applications (mainly using it in combination with other tools)

2. Working Methodology

Let's start the presentation with a historical step by step approximation. There are many users that start having a piece of code that works well enough. After plenty of struggling against the problem and the compiler, there is something ready to be shown to your instructor/manager. Of course, nobody will accept that nasty looking pile of statements, with very peculiar indentation criteria. Then rises the need to normalize the source code for other people to be able to read it. OK, go and use a pretty printer. Readers will appreciate your code a little bit more.

Very quickly the need for more complete information arises. It is not enough to have an easy to read program. You need to inform potential readers of your design choices, problems and solutions, efficiency constraints, weaknesses, and so on. You need to add natural language explanations. No problem, you have a nice syntax for `(* embedded comments *)`. If you are careful enough, you'll come with a piece of thoroughly commented code. Your instructor/manager will start appreciating your work, and accept passing it into the maintenance phase.

If you are really willing to provide a document, part of which is the code of your program, you'll quickly look for using a text processor in order to typeset a high quality document. Of course, there is always the opportunity of having separate documentation. One file for the code, another for the natural language explanations. Actual experience shows that this structure is hard to maintain, and error prone. You have to edit two different files simultaneously. Thus, the two files model is rejected.

Accepting that we want to keep together code and documentation, there are some new problems. If you are lucky the programming language will interact with your text processor, and you can embed text processor commands as comments. There will be some noise with hanging `(*` and `*)` (for instance, if using Pascal), but you get something that can be sent to a laser printer. Maybe you are less lucky. For instance, if your programming language is *Ada*, it will require one comment per line starting with two hyphens. If your text processing language is *troff*, it will require one command per line starting with a dot. Clearly incompatible requirements.

In these situations you have to invent your own criteria and use some tool to make the compiler and the formatter compatible. For instance, you may use a simple *sed* filter to drop every line starting with a dot before passing the file to the compiler.

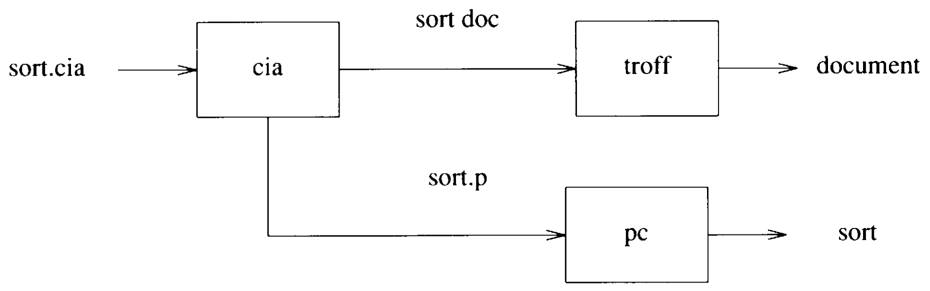
Still now, there are more subtle problems. It happens that there is a very important procedure in line 1256. You would like to move it to the very first sections of your document, but the target language scope rules forbid it. And the other way round, you would like to send the first one thousand lines of house-keeping procedures, back to the seventh appendix. But, once again, the scope rules are too rigid.

If your target language accepts modules, you may run into new problems. You have separate files, but between some of them the relationship is so strong that you would really want to document them altogether. It's likely that the maintainer will greatly appreciate this kind of documentation clustering.

Conclusion: you need something that permits intermixing programming statements for a language processor and formatting orders for a text processor. Even more, you need something flexible enough as to escape rigid ordering of the pieces of software.

We think that many programmers have found themselves in the previous story. It's really our own situation. And when we think back over it, we have the strong feeling that there is something very basic that is wrong. More concretely, programmers should not start having a bunch of statements that run. Just the other way run, programmers should start with a textual description of the system that they have to make run. Once you have the document, you start refining it, partly adding more explanations, new algorithms, reporting of errors, possible efficiency bottlenecks, etc, partly adding actual code. In other words, design should precede coding, design reasoning should be stated on a document before there exist the implementation statements.

To end this part, let us present a trivial, yet prototypical example. It will permit a first approach to the pieces we are dealing with. CIA is the name of the tool we are about to present. Let's have a sorting program `sort.cia` partly written for *troff*, partly for *pascal*.



To get the document, you filter it through the CIA and the text processor

```
...> cia sort.cia | troff -ms
cia: -> sort.p
```

As a side effect, a Pascal file `sort.p` is produced, that is processed as usual

```
pc -O -o sort sort.p
```

3. Design Criteria

Main desire is to have a tool to write programs that are documents and/or write documents out of which programs may be extracted. Code and text must be kept together for editing, and will be split apart with the help of a tool. The tool must permit flexible documentation of program files, and keep coherence between files.

The tool we are using is named CIA. It is just a pun on the very well known U.S. Agency. We chose the name to reflect that the tool is able to cope with many files keeping information coherent. From other point of view, the tool is able to extract a whole history starting from a seed and pulling out a thread of reasoning. That idea is what D.E. Knuth calls *tangling*. As a final consideration, the name is short and easy to remember.

The CIA accepts an input file that is basically text oriented, embedding text processing commands. In this file, pieces of code may be *named*, and be referred to from other parts of the file. Input files include CIA commands that are interpreted for the extraction of the code in the proper order. The CIA has been developed in a UNIX environment with standard text processors in mind, e.g. *nroff*, *troff*, TEX, LATEX [Lamport, 86], etc. The CIA is independent of the target language, e.g. C, Pascal, Modula-2, Ada, shell scripts, makefiles, other CIA files, etc.

In UNIX jargon, the input file (or the concatenation of several input files) is filtered onto the *standard output*, processing only those lines that are CIA commands. The standard output is the basic document output.

Other files may be generated as side-effect. The overall idea is to mark pieces of text being able to recall them as many times as you wish, in any order you like. Thus, you can write a program intermixed with explanations, top down or bottom up, as you think it is more readable, and then dump it in the peculiar format required by the corresponding target language processor.

The CIA is line oriented. That means that CIA commands cannot be interspersed with the rest, but must be in lines of themselves. Line breaks in the source file are preserved in the generated files. That means too that no attempt is made to pretty print anything. A Pascal program will be output with the same line breaks, and the same relative indentation as in the source file. This decision has several benefits. First, it makes the tool language independent. You may still pretty print the code using any pretty printer you have around. If you have languages with fussy syntax, say makefiles, you may easily cope with them, nobody changing your tabs and so. Usually the code will be enriched with information to permit the compiler to refer to the correct line in the original file when an error is detected. For instance, for most UNIX processors, `#line` lines. But not every language permits this information to be added (e.g. *yacc*, *lex*, *shell*, *make*, $\cdots$). In this cases you need to read the generated file and then go to the original file for making corrections. It is clumsy, but possible.

4. Features

This section presents the kind of commands the CIA will accept. It is a simplification of the *user's manual* [Mãnas, 87] that just pretends to provide the flavour of the tool.

4.1. Basic Definitions

Pieces of code are put together and given a name by means of a couple of bracketing commands

```
^define [count] key
        ... body ...
^end
```

The *key* is any string of characters up to the end of line. The optional count is the expected number of times this definition will be used. Default is 1, and a warning is emitted if the actual number of instantiations doesn't agree with that count. Main use of this feature is to check that every definition is used somewhere, unless it is explicitly stated otherwise.

In order to *use* a definition, there is a replace command

```
^replace [indent] key
```

The interpretation of the replace command differs upon being generating the standard output or a secondary file. For the standard output, the definition is not expanded, but there will be a line showing in some format selected by the user, that a replacement is to be made there. On the other hand, while outputting to a secondary file, the body of the corresponding definition is expanded on line.

The optional indentation parameter specifies a relative (`[+|-]number`) or absolute (`number`) indentation of the body to be expanded. The number is measured in tty-character units. Relative indentation is done with respect to the last non-empty line read from the source file. Command lines are not considered for indentation measures. Absolute indentation is done with respect to the left margin. The first line of the body of the corresponding definition is indented so much. The other lines of the body preserve their indentation relative to the first line. Thus, the body is shifted as a block. If no indentation is provided, the definition is expanded as it was written, without any shifting.

Secondary files are generated as side-effects

```
^file filename
      ... body ...
^end
```

The *body* is copied onto the specified file. Usually, `^replace` commands will appear in the *body* forcing the unthreading (or tangling) of the code.

4.2. Output Fine Adjustment

Usually, outputs generated by means of `^file` commands are just written onto the specified file. The only extra processing the CIA will do is checking whether the output is a modification of an already existing file. That is, when a file is to be written, the CIA will check if it already exists. If so, it will check whether they are equal. If both checks succeed, the file isn't actually overwritten. This trick stops *make* from regarding the file as modified, stopping any further processing. This feature is usually required when several files are generated as a result of running the CIA. Modifications may not affect to all of them, and *make* should be instructed about actual modifications, not only rewrittings.

Furthermore, the output may be send to a shell as standard input. The CIA will not care the result of the execution of the shell, whatever it may be. There are many uses of this feature. First, as an example, the basic behaviour may be emulated as

```
^file | cat > /tmp/cia$$; `
cmp -s /tmp/cia$$ filename || cp /tmp/cia$$ filename;
rm /tmp/cia$$
```

As another usual situation, let's consider the case of using *vgrind* to pretty printing the code. While in the standard output you wish to have `.vS` `.vE` commands, these should not go to the code file. Deleting these controls is pretty simple

```
^file | sed '/^.v[SE]$/d' > filename
```

Other frequent applications are handling of text processor escape characters (using *sed* once again), sending mail, sending to a printer, etc.

Name expansion of the shell may be fruitfully applied as in the following typical examples,

```
^file | cat ~/filename
^file | /lib/cpp -DOPTION > $PROJECT/src/file.c
^file | rsh host cat '>' '~pepe/file'
```

4.3. Name Substitution

It has proved very convenient to have access to the name of the source file in several formats.

key	*means*	*example*
%F	full pathname	/usr2/pepe/src/file.c
%Z	path	/usr2/pepe/src/
%N	full name	file.c
%P	prefix	file
%S	suffix	.c
%%	percent	%

This permits having standard patterns as CIA files. Eg.

```
...> cat ~someone/file.cia
...
^define code for file %P.a
...
^end
...
^file %Z%P.a
^replace code for file %P.a
^end
...
```

Further flexibility may be achieved by running a macroprocessor before the CIA, eg. *m4* or *cpp* ,

```
...> /lib/cpp -DOUTFILE=/usr/et/earth
```

4.4. Tailoring

Most CIA commands are given a visible representation in the standard output. In order to make the tool independent of any text formatter, the default printing may be customized. Let's start with the most common case: a source command line as

```
^define this is a sample definition
```

is echoed to the standard output as a line

```
<< 24. this is a sample definition >>:=
```

The number is automatically provided by the cia reflecting the list of definitions as they appear in the source file.

This default format is specified as

```
^fdefine << %D %K >>:=%n
```

where `%D` stands for the definition number, `%K` for the key, and `%n` for a carriage return. `%R` is also accessible as a list of numbers of definitions and files that use this definition.

If you are using *troff*, you may wish to get finer control of the layout. One of the easiest forms is to generate a macro

```
^fdefine .^D "%D" "%K" "%R"%n
```

And the macro `^D` shall be defined somewhere else, either being present in the source file for the CIA, or loaded from a macro file (via `.so` commands), or any other procedure. The definition of the macro is up to the user, and will usually refer to other macros for direct access to standard packages (e.g. *ms*). A frequent application is to embed in the macro the required commands to generate a *table of definitions* at the end of the document. Standard packages as *-mm* may greatly simplify it. Here is a simple definition that provides an *-ms display*,

```
.de ^D
.DS L
<<   >>:=

..
```

Usually, the `^fdefine` command is used to generate a macro according to the intended text processor conventions, macro that is later on expanded. But, of course, the CIA may expand more complex formats. As an example, consider the case of *vgrind* that must be run before *troff*, implying that `.vS` `.vE` commands cannot be the result of expanding a *troff* macro. A nice format for *troff* and *vgrind*, without cluttering the files, may be achieved as

```
^fdefine .DS L%n<< %D %K >>:=%n%n.vS%n
```

A similar situation will arise around `^end` commands,

```
^fend .vE%n%n<< end of '%D %K' >>%n[[ used in %R ]]%n.DE
```

Same format tailoring may be applied to `^replace` commands (default `"<< %D %K >>%n"`), and to `^file` (default `"empty"`).

These formats control the printing on the standard output. There is another command for controlling printing on the secondary files. Here the problem isn't pretty printing or text processor controlling, but keeping track of positioning of the pieces of code in the source file. By default, when a file is dumped, there is no information about the structure of `^replace` commands that generated it. Most languages in UNIX accept `#line` directives to keep track or original source lines. The CIA may optionally provide this link. But other language processors do not. In either case, it is sometimes desirable to have that information for the case the generated file must be read by humans. The trick uses to be writing a comment. For instance, if the target language is C,

```
^infile (* %D %K *)%n
```

Or in *sh* files,

```
^infile # %D %K%n
```

4.5. Miscellaneous

A `^copy key` command is provided to *emulate* the definition of a key. It is usually applied to the production of slides from a document. In the document there are `^define` commands. These generate a nice output for the standard output. Then you wish to generate a secondary file that contains a copy of those definitions intermixed with text processing commands to get big letters and all the stuff for slides. Since you are not normally willing to repeat the `^define`, nor allowed by the CIA, nor willing a `^replace` that would unwind the nested structure, you may use the `^copy` and get the *effect* of having the `^define` again.

Another useful command is one that prints on the standard output the skeleton of a nested structure of definitions. This is better explained by means of an example. Where in the source file you have `^skeleton key`, you'll get an output as

```
<< 1. Entire program >>
  << 2. Global constants >>
  << 3. Global variables >>
    << 6. Global variables 2 >>
    << 8. Global variables 3 >>
    << 12. Global variables 3 >>
  << 4. The random number generation procedure >>
  << 5. The main program >>
    << 7. Establish the values of M and N >>
    << 9. Initialize set S to empty >>
    << 10. If T is not in S, insert it and increase size >>
      << 11. Insert T into the ordered hash table >>
    << 13. Print the elements of S in sorted order >>
```

The example is inspired in that of D.E. Knuth [Bentley, 86].

This completes a sharp presentation of the features of the CIA. There are some more commands for file inclusion, comments, and that's all.

4.6. What the CIA doesn't do

When designing a language it is as important what it provides and what it doesn't provide. Of course, there are plenty of features that it doesn't provide and nobody will reasonably complain about. But there are a few features that are more or less frequently required and, despite users request, we have left out. We have seriously tried to make the CIA as simple as possible. If there is a UNIX tool that provides some facility, use it in connection to the CIA rather than providing it again. Tools must combine. Repeated work is not only tedious, but error prone and harder to maintain.

Most frequent request is for typical macro language commands as text replacement, conditional expansion, and so on. The answer is given in two steps. First, *which macro processor would you like the CIA* to look like? Second, *ok, go and use it as a filter before or behind the cia*! Personally, we are used to *cpp* and do extensively use it either before or behind or, with some care, even twice.

Sometimes we get requests for accessing environment variables as *sh* or *csh* do. Usually, the syntax for `^file` permits passing control to the shell, and nothing more is required. Alternatively, you may use *make* to instantiate the CIA and write things as

```
/lib/cpp -DOBJECT=$VAR file | cia | troff -ms

echo "define(OBJECT,$VAR)" | m4 - file | cia | troff -ms
```

There is a feature of *web-tangle* that is sometimes requested for the CIA, namely the facility for extending a `^define`, what in *web* appears as "+≡". That seems to be a very *Pascal* oriented feature for coping with pseudo-global variables. Our point of view, and not everybody agrees, is that a `^define` is both a piece of code and a proper name. As a piece of code, it may be acceptable to extend it later on, although there are two counter arguments. First, if the main CIA activity is combining via `^replace`, why do we need another notation? Second, in which order do extensions combine? Probably in the same order as they appear in the source file, but that's plainly against CIA conventions of being ordering independent. There may be problems if the ordering of the definitions is modified (e.g. defining a variable of a *type* that is not yet defined), and you may easily modify the ordering by, for instance, moving a few definitions onto an appendix.

As a proper name, we would strongly force users to put the name and the code together and not splitting it over. Editing of a split specification is hard if you do not have a multiwindow editor and you have to navigate around to find each piece.

For the case of *Pascal* pseudo-global definitions, there are much better solutions as preparing a definition for each block of variables, and combine them by means of `^replaces`. The result is better documented, since there is a proper name for each proper thing, and it is more robust, since permits you to move the definitions around.

5. Sample Applications

The CIA was first designed upon a wonderful presentation in the Communications of the ACM in June 1986 [Bentley, 86]. We found the ideas extremely useful for our actual problems in software teaching and development, and started the production of a first implementation. It took less than one month. Later on (about February 1987) we got the other main paper on the subject [Knuth, 84] We kept on with our tool that was starting to evolve upon wide experience within the department. Lastly, early in 1988, we have the Donald Knuth's tools *web* and *tangle*, but still small experience with them.

Our experience with the CIA starts with prototypical cases of Pascal programs. There is one source file that produces the document and a Pascal file as a side effect. That case was presented in the introduction and doesn't deserve more attention.

5.1. Multimodule

There is more fun when you come into C language and wish to organize your application as a set of files. Sometimes, you wish different files just for reasons of object scopes. Then you may wish to use the CIA to generate them by means of two `^file` commands, while keeping only one source file and a common documentation. It is likely that you will wish to have the building specification (the *makefile*) as an appendix of the common document.

If the target language is *Modula-2* or *Ada*, then every piece of software is split into a *public interface* and a *private implementation*. If the project is small, you may wish to keep both together in order to have a common up-to-date documentation, and preserving coherence by sharing the same `^defines`.

If the project grows bigger and several people become involved, then software engineering practices will change the file organization. Several interfaces may be documented altogether with the architectural design of the application, and the implementations will be documented separately. Still coherence may be achieved by including the interface `^defines` in the implementation documentation. Once again, there may be a *makefile* described at and generated from the Architectural Design Document.

5.2. Small Changes

More sophisticated use permits generating slightly different versions of the same program. That is a very usual situation in teaching. You write a program and then start applying small variations. You would like to have a single document for everything, and yet have executable code for each of them. A similar situation arises when generating code for different target machines.

Here there are two problems. The first one, generating several files, is simply resolved by using several `^file` commands. The second one is getting the correct pieces into each one. If all the definitions are made at the same level, it is a children game to build the correct puzzle for every file. But if we wish to distinguish the versions by changing some deeply nested definition without changing the common environment, then it is not so easy.

Let's work on a small example. We have two slightly different definitions

```
^define version 1: code        ^define version 2: code
   ... body 1 ...                   ... body 2 ...
^end                            ^end
```

That are used in a common definition. In order to distinguish, we shall use the *cpp* preprocessor. Of course, any other tool could be used (e.g. *m4*, *awk*, etc.)

```
Main document:                   Appendix:
  ^define common definition        ^define version #: code
   ...                             #ifdef VERSION1
   ^replace version #: code        ^replace version 1: code
   ...                             #endif
   ^end                            #ifdef VERSION2
                                   ^replace version 2: code
                                   #endif
                                   ^end
```

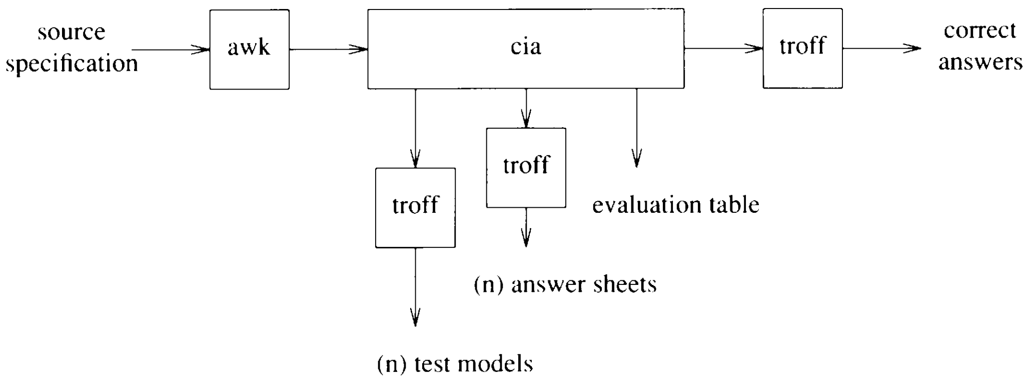
When the actual files are output, one or the other version is chosen

```
^file | /lib/cpp -DVERSION1 > version1.c
^replace common definition
^end
^file | /lib/cpp -DVERSION2 > version2.c
^replace common definition
^end
```

There are many ways of getting the desired effect. Here we just intend to show the easy interaction with other UNIX tools.

5.3. Many Coherent Outputs

Another powerful application involving different filters is test generation. In teaching environments we often have the need of producing tests for students examination. A nice tool may be developed that generates from a single source a set of test questionnaires, answer sheets, test documentation, etc. For the same exam we wish to have several permutations of the same answers for the same question. Each student receives one model and an answer sheet for that model. When they leave the examination room, there shall be a document showing the correct answer to each question, and the (positive or negative) weight of each choice. Thus, students may fastly evaluate themselves. There shall be too a short file relating each answer and each weight for each model. This last file shall be read by an automated correction tool.



The single source contains single command lines to start questions, and answers with its corresponding weights. A simple *awk* program is enough to translate questions and answers into `^defines`, deleting the weight from them. Lastly, a standard collection of `^file` and `^replace` commands is appended, yielding a random permutation of answers. Thus, the CIA will output all the necessary files, with the same answers but different ordering.

5.4. As A Macroprocessor

The CIA may be seen as a specialized macro processor. Pieces of code may be given a name and then expanded on line in several places. Run time efficiency increases without function call overhead. While, source code is still split within small chunks of information. In other words, it is not required to invent procedures for documentation purposes: just invent `^definitions`.

Some methodologic considerations should be followed by users. It has proved a good criterion to write *bodies* of about 12 lines. Giving a name to smaller pieces of code uses to introduce too many `^defines`. On the other hand, `^defines` should be smaller than a normal screen (24 lines) for on-line editing. Nevertheless, 20 lines of code will surely deserve a name by themselves.

Most users will naturally provide *syntactic definitions*. That means that the same `^define` will encompass both the `begin` and the `end`. The alternative is to start a block in one `^define` and finish it in another. We strongly recommend the first method. You keep syntax brackets together and then refine the contents in another `^define`. Users that do not follow this thumb rule use to produce extremely confusing documents.

5.5. Make Integration

Makefiles may be easily adapted to automatically operate on CIA based applications. The default rules for file production must be extended in the natural manner. Here follow a few of them.

```
.SUFFIXES:      .cia

.cia.c:
        cia $(CIAFLAGS) $*.cia > $*.doc

.cia.o:
        cia $(CIAFLAGS) $*.cia > $*.doc
        $(CC) $(CFLAGS) -c $*.c

.cia:
        cia $(CIAFLAGS) $*.cia > $*.doc
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $*.c

.cia.doc:
        cia $(CIAFLAGS) $*.cia > $*.doc
```

5.6. Source Code Distribution

Lastly, we would like to mention the opportunity of distributing undocumented source code. That is the case when the distributor doesn't have access to the target machine in order to compile the code at home and distribute just the objects. Since he is forced to send out sources, the CIA may be used to distributed fully undocumented code, in fact useless but for remote compiling. At the same time, a fully documented source is kept at home for maintenance and further releases.

6. Related Tools

There is a tool clearly related to ours. It is the *web-tangle* package from D.E. Knuth. We shall shortly highlight its differences.

Perhaps the most fundamental difference is in objectives. While Knuth's tools are intended for high quality typesetting of Pascal programs, the CIA is intended for documenting any code. Both tools may productively interact. The wonderful Pascal pretty printer embedded in *web* may still be used on the output of the CIA for high quality typesetting of Pascal programs. Surely you'll need filtering out @-commands, but that's easy. Even, the *change file* facility may be used to generate customized files to be processed by the CIA, although a simple program for *awk* would normally suffice. Personally, we would prefer the classic *cpp* conditional compiling. And there finishes the interaction.

Another deep difference may be found in the role of the tools in a UNIX environment. While *web* and *tangle* are self-sufficient tools, embedding a macro processor, a pretty printer, and a peculiar version of Pascal; the CIA is expected to work as one filter more, thus simplifying its behaviour. Whatever another tool may do, let it do it!

From a user point of view, we have our own experience. After a small introduction people start using the CIA very quickly. Say one hour. We provide some standard macro packages either for configuring the CIA (`^fdefine` commands, for instance), and for customizing the text processor (usually *troff* or LaTeX macros). Of course, the user must know of the target language and of the text processor.

We lack of a similar experience with Knuth's tools, but we are distressed by a phrase of the *web* user's manual: "··· WEB users must be highly qualified, but they can get some satisfaction and perhaps a special feeling of accomplishment when they have successfully created a software system with this method.". We have actually appreciated a quick satisfaction on CIA users.

7. References

[Bentley, 86]
J. Bentley, D.E. Knuth, and D.McIlroy. *Programming Pearls: A Literate Program*. Comm. ACM, 29(6), pp. 471-483, June 1986.

[Knuth, 84]
D.E. Knuth. *Literate Programming*. The Computer Journal, 27(2), pp. 97-111, May 1984.

[Knuth, 84a]
D.E. Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1984.

[Lamport, 86]
L. Lamport. *LATEX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, 1986.

[Mañas, 87]
J.A. Mañas. *CIA manual*. Dpt. Telemática, June, 1986.

Multilevel Security with Fewer Fetters

M. D. McIlroy

J. A. Reeds

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We have built an experimental UNIX system that provides security labels (document classifications), where the security labels are calculated dynamically at the granularity of kernel activity, namely, at each data transfer between files and processes. Labels follow data through the system and maintain the lowest possible classification level consistent with the requirement that the labels of outputs dominate the labels of inputs from which they were computed. More rigid control is exerted over the labels of data passing out of reach of the system to and from tapes, communication lines, terminals, and the like. Necessary exceptions to the security rules (as for system administration, user authentication, or document declassification) are handled by a simple, but general, privilege mechanism that can restrict the exceptions to trusted programs run by ``licensed'' users. Privileges are subdivided; there is no omnipotent superuser. Carefully arranged data structures and checking algorithms accomplish this fine-grained security control at a cost of only a few percent in running time.

Dynamic labels should help mitigate the suffocating tendencies of multilevel security. At the same time dynamic labels admit covert channels by which dishonest, but authorized, users can leak data to unauthorized places at modest rates. The system is still highly resistant to other kinds of threat: intrusion, corruption of data by unauthorized users, Trojan horses, administrative mistakes, and joyriding superusers. In most real settings, we believe, worries about potential leaks will be far outweighed by these latter concerns and by the overriding consideration of utility.

The standard security mechanisms of the UNIX system are, in military parlance, ``discretionary'': protection depends primarily upon the individual owners of data taking care to set permissions on files. Some automatic help is offered. the owner/group mechanism, *umask*, and clean files uncontaminated by old trash from shared disks. The responsibility for further precautions, such as setting owner-only permissions on files in the shared temporary directory, is delegated to programs.

A carefully administered UNIX system can be quite resistant to penetration. But careful administration is not easy. Great reliance is placed on the probity, accuracy, and vigilance of superusers. It is all too easy for a busy superuser inadvertently to misset permission bits, to execute a Trojan horse[1], to make temporarily unprotected copies of secrets, or to promote unvetted files to trusted status.

Because UNIX systems are simple enough to be administered by amateurs, whose first interest is in use, not operation, security holes are the rule, not the exception, in real life. A few famous security holes have been distributed in major software. And some add-ons, such as Berkeley's network file system, seem to have been deliberately designed for insecurity.

Is there, then, hope of running a UNIX system securely? Of course. Lots of systems are run quite soundly today. Raising the question a notch, is there hope of making a UNIX system that conforms to government-style security policies?

[1] The hoariest of all: a bad guy feigns trouble, and asks a superuser for help. The first thing the superuser does is `cd badguy; ls`. The game is up. The bad guy's own program named `ls` has been executed by the superuser; it has silently bugged some setuid root program, removed itself, and then called the real `ls`.

In a sense the answer is yes; UNIX systems need less fundamental patching than most to achieve government security goals. But the yes must be qualified. In almost any production system it is difficult, if not impossible, to guarantee that many megabytes of code do not contain one fatal crack that can topple the whole edifice. Such guarantees are especially elusive in environments that change frequently, as do so many typical UNIX installations. How can one be sure that no flaws are inserted by any of the software tools—shell, editor, compiler, assembler, library—that touch new code as it is being installed?

In another sense the answer is not so clear. Rigid security measures must surely damage the plasticity that attracts users to the UNIX system. Is the conflict so fundamental that a ''secured'' system will lose its appeal? The possibility is very real under ordinary security models; hence we have undertaken a somewhat more flexible approach.

We have built an experimental system with mandatory controls: security classification automatically follows data through the system. At the same time we have blunted the vulnerability of the system to mistakes by the superuser. In our system all data files have security classification *labels*, with ''higher'' labels designating more sensitive data. The normal flow of data must be *up*: output labels must in general dominate input labels. A security system also needs escape hatches for declassifying data that is no longer sensitive, or extracting nonsensitive parts from sensitive documents. For this purpose we allow certain carefully designed *trusted* programs to violate the rules and produce output with labels lower than input.

We have attempted to provide mandatory controls without utterly destroying the basic feel and productivity of the system. To obtain early warnings of snags caused by the controls, we are doing our development work under the system itself. At the time of writing we are working quite honestly and relatively comfortably within the confines of the system.

The main idea

Each file or process has a label, shared by all data in it[2]. Terminals and other devices such as tapes have labels that reflect the system's understanding of the clearance of the source to which the device is currently connected. The labels form—almost—a mathematical lattice. Whenever a system call causes a transfer of data, the labels are checked to ensure that data flows only up the lattice.

The security of data explicitly passed among labeled entities is safeguarded. Examples of protected transfers are bytes transmitted by *read* and *write* and bits set by *chmod*. Implicitly set inode data, such as file modification times and link counts, are also protected as far as possible without making the system unusable.

Other ways of communicating information, including but not limited to arguments of *exec*, error returns from system calls, file access times, the identity of open files, and otherwise inferred knowledge, we declare to be ''covert channels.'' We have studied covert channels and arranged to throttle or stop completely covert channels of significant bandwidth. In effect we have divided information transfers into ''lawful'' transfers, which honor the US Department of Defense ''Orange Book''[3], and covert channels. Just which covert channels to leave unplugged we have decided by balancing risk versus utility and compatibility.

We attempt to minimize ''label inflation'' by keeping all processes and files at their minimum allowable labels as long as possible. The label of a process will increase only when necessary and only as far as needed to allow reading of inputs. Similarly when the label of a process exceeds the labels of its output files, the file labels will rise.

A few system programs must be exempt from the usual label checking. Such programs are granted special privileges—for instance to set the label on a user's terminal at login time, to read foreign tapes, or to perform backups. These privileges are zealously guarded: no program can pass its privileges intact to another, alter a privileged program in any way (aside from removal of privilege).

Thus we have three kinds of security mechanism in our system: (1) the usual UNIX permission scheme, based on userid and groupid and the familiar `rwxrwxrwx` bits, but with the superuser stripped of the

[2] For technical reasons, seek pointers also have labels of their own. Seek pointers are shared between processes; information can flow through a shared seek pointer (via *lseek*) at a substantial rate—thousands of bits per second. Since a seek pointer is ''written'' into by *read* as well as by *write*, the contents of a seek pointer, unlike the regular contents of a file open for reading, must have a label as high as that of the reading process. Hence the separate label.

[3] *Department of Defense Trusted Computer System Evaluation Criteria*, Department of Defense Computer Security Center, Fort Meade, MD, 15 August 1983. This bible has set the terms of discussion for most current work in computer security.

right to ignore permissions, (2) the label scheme, which strives only to maintain correct label relationships, and which pays no attention at all to userid or groupid, not even superuser, and (3) the privilege scheme, which guards the administration of labels and of the privilege scheme itself.

Labels

A label can be any element of a given finite lattice. In addition there are two nonlattice labels, **NO** and **YES**. No data may flow to or from a file or inode labeled **NO**; it is effectively blocked out of the system, and can only be readmitted by special arrangement. "External media," such as terminals, tape drives, and raw disks, where labeling is beyond the control of the usual mechanism, are normally marked **NO**. Label **YES**, on the other hand, is universally permissive. Only one file, `/dev/null`, is marked **YES**. At the moment no other file can gain such blessing, but it might also be appropriate for an append-only audit file.

Our lattice is the lattice of subsets of 480 items, represented by 60-byte bit vectors. How these bits are used is arbitrary. For example, the first three bits might represent the customary classification levels—unclassified, confidential, secret, top secret—encoded as 000, 001, 011, 111 respectively. Further bits might represent compartments: 000 100 for Iran, 000 010 for Nicaragua, etc. Oliver North would have been cleared for 111 110. A possible history of a process initially labeled secret (011 000) is:

> Create a new file `north/contragate`; it is labeled (000 000) by default, but writing in directory `north` causes the label of `north` to become at least secret (011 000).
> Read `iran.data`, which, say, is confidential and compartmented (001 100). The process label rises to (011 000)$\cup$(001 100) = (011 100).
> Read `nicaragua.data`, top secret and compartmented (111 010). The process label rises again to (111 110).
> Write `north/contragate`. The file label rises to (111 110). The directory label is unchanged.

Not all labels can change automatically. A label may be "frozen", which stops operations that would normally require a label change. In particular, labels of terminals are guaranteed to be frozen, typically at the value determined by login. Suppose our example process had been initiated from a terminal that had been cleared only for top secret Iran data (111 100) and attempted finally to write to the terminal. The write would fail, thus keeping Nicaragua data from a user not known to be cleared for it. Further attempts to launder the label, perhaps through a pipeline like `cat north/contragate | grep .`, would meet the same fate. Only a properly authenticated fresh login (or subsession) can authorize the terminal for the higher label.

The idea of a lattice of labels is well known. Our deviation from the strict model, with **NO** and **YES**, answers needs to regulate entry from places where labels are not under control of the system, and to deal with the important special case of `/dev/null`.

Privileges

Our privilege mechanism is simple, but flexible. It allows policies of placing trust solely in particular programs or solely in particular users. It also allows much stronger policies that restrict special powers to trusted users using trusted tools.

To some extent the privilege mechanism may be understood as partitioning the supreme powers once accorded to the superuser. Superuser status itself is diminished. The superuser is fully bound by security labels and cannot ignore write permissions. Largely to avoid rewriting masses of code, the superuser retains most other powers. Thus the superuser can still do damage (to data he is cleared for), but mainly by tedious methods that leave tracks—changing modes and owners. Superuser status must be augmented by privilege to execute powerful restricted system calls such as setting the userid or mounting a file system.

We have identified five distinct privileges, each governed by one-bit *licenses* and *capabilities,* which are separate from labels. A trusted process or file is one with nontrivial capabilities or licenses. In the strictest policy regime each privilege of a process p is determined by the intersection of the process's license for that privilege and the capability for that privilege of the program file f it is executing:

$$Priv(p) = Lic(p) \cap Cap(f).$$

Process licenses are assigned at login, are inherited across *exec,* and may be relinquished at will, never to be regained. Licenses effectively identify trusted users, while capabilities identify trusted programs.

For more liberal policies, it is possible to grant a default "system capability", $Cap(s)$, to every file by the rule

$$Priv(p) = Lic(p) \cap (Cap(f) \cup Cap(s)).$$

By setting $Cap(s) = \textbf{true}$, we can make $Priv(p) = Lic(p)$, which means that any program to do magic provided its user is licensed. In such a regime a superuser possessing licenses for all privileges could act with the same impunity as a standard superuser.

It is also possible to give a program file a license, $Lic(f)$, making the program "self-licensing" for one or more of its capabilities. Then the effective license of a process p executing program f is $Lic(p) \cup Lic(f)$. Self-licensing is limited by another policy constant, the "system license", $Lic(s)$, which is used as a mask. The full formula for determining each privilege of a process is

$$Priv(p) = (Lic(p) \cup (Lic(f) \cap Lic(s))) \cap (Cap(f) \cup Cap(s)).$$

In a typical self-licensing case, where $Cap(s) = \textbf{false}$, $Lic(f) = Cap(f)$, and $Lic(s) = \textbf{true}$, this reduces to $Priv(p) = Cap(f)$. In this regime a self-licensed program gets power in much the same way as does a setuid-root program in standard systems, except that the power is not inherited across *exec*.

With appropriate settings of the two (compile-time) system policy constants, $Lic(s)$ and $Cap(s)$, our privilege model is able to mimic the disparate privilege features of most current operating systems. In our experimental system we have set $Cap(s) = \textbf{false}$ for every privilege. We have also also set $Lic(s) = \textbf{false}$ for the most powerful privilege, "set privileges". Thus privileges can be set only by trusted users using trusted programs.

We have identified five privileges:

Mount. The right to make new data sources or sinks available to the system.· One way is by changing a file label away from **NO**; a second is by the *mount* system call; a third is by changing the label on an external medium. A process with mount privilege would normally execute an authentication protocol before actually performing any of these operations.

Nocheck. The right to read or write data without regard to security label (but still respecting the standard permission scheme). Although mount and nocheck both provide extraordinary access to data, they are qualitatively different. Nocheck handles (and may censor) every suspect bit. Mount opens resources to the whole system—a much more sensitive responsibility.

Set licenses. The right to increase the license or ceiling of a process. The principle use for this is in setting up "sessions", where a user entitled to play more than one role wishes to suspend one role temporarily and switch to another. Sessions are merely a refinement of `su`, which changes rights by the crude expedient of changing identity.

Set privileges. The right to change file capabilities and licenses. We expect not more than one or two programs to be given this most powerful of all capabilities. In a thoroughly security-conscious installation, only an identified security administrator, different from the system administrator, would be licensed to set privilege.

Write uarea. The right to change values, such as userid, that are remembered by the system for the benefit of the process and its offspring. This peculiar capability arises because a child process need not be as highly classified as its parent. Without some control, uarea items (especially BSD group permissions) would provide a covert channel of significant bandwidth.

By dividing privileges we promote safety from errors by an omnipotent superuser. At the same time we introduce complexity, which can cut the other way. Thus we have deliberately kept the number of identified privileges small. We have refrained from defining new special roles (for example system administrator, operator, or security administrator) in the superuser tradition. Notions of such roles did influence our choice of privileges and will guide the design of administrators' tools. But the notions seem inappropriate to build in at the ground level: no single administrative model makes sense across the spectrum of real installations.

System features

To implement the above facilities relatively few new system features are involved:

New system calls get and set file labels. Another new system call sets the process label. Privileges and frozenness are set along with labels. Unless executed by a trusted process, the system calls permit only safe changes: labels may not decrease; process privileges may not increase; file privileges may not be changed.

A special system call allows nocheck processes to confine their powers to certain files. For example, consider `df`, which needs nocheck privilege to read the file system device. Its outputs, however, should be subject to ordinary security checks to prevent a mole from getting his message through[4].

Every process has an inherited *ceiling* label, above which the process cannot do any business. This has little to do with stopping ordinary leaks: if a lowly process raises its label high, its output will be high and thus protected anyway. It does, however, cut off some possibilities for mischief with covert channels. And it prevents unauthorized userids from injecting noise in high places.

Mounted file systems also have ceilings, both on labels and privileges. File system ceilings may be used to restrict the content of file systems being prepared for export, or to prevent contamination, especially by unknown privileged files, from imported file systems.

A directory may be "blinded." Untrusted processes cannot open a blind directory for reading, and every new file created in such a directory is assigned a random name. A new system call retrieves the name. Blind directories are immune to automatic label changes and thus provide a convenient way to gather, yet keep hidden, data of disparate labels, as for the temporary directory `/tmp`.

Implementation

Dynamic label changing involves considerable overhead of implementation. It is insufficient simply to add label checks at file open. In principle, labels must be checked on every read, every directory search, and every write, including writes of new entries into directories. When a write check fails, the file label is raised if possible; for a read the process label is raised. Every other process dealing with the file must become aware of the change on a fine time scale; in the worst case a label may change between disk blocks of a long IO transaction. A carefully designed data structure for intra- and inter-process notification of label changes has accomplished this with only a few percent time overhead.

Space overhead is another matter. A production-size kernel is considerably bigger than before: about 16K of extra text and nearly 400K extra data for a 500-process system. In partial compensation, uareas are smaller. To accommodate labels, inodes on disk have been doubled to 128 bytes.

In effect labels flow along with data. Upon *exec* a process begins with the lowest label possible: the least label that dominates both that of the executed file and that of the arguments. The arguments, of course, have the label of the parent process. However, if no arguments are supplied, as for an ordinary filter, the argument label is taken to be the minimum, or bottom element of the lattice. Thereafter the label of a process changes to keep up with the data that it reads. (Notice that the *open* system call does not read; *stat* does.) In particular labels may propagate through pipes.

Similarly files are created with the bottom label. (We accept a narrow covert channel through the mode field.) However, the label of the directory in which a new file's name is recorded must dominate that of the creator; the name could bear secrets.

Covert channels

Having classified many communication paths as "covert channels," we have an obligation to recognize generic classes of covert channels and to characterize their effectiveness. This we have done. Aside from very narrow "timing channels," most of the covert channels in our system involve unusual behavior: forking enormous numbers of processes or opening enormous numbers of files[5]. Thus any extensive use

4 If `df` is exempt from all security checks, the mole can get a message to the standard output this way:

```
df /dev/disk0 /dev/disk1 /dev/disk1 >unclassified
```

which produces binary code in the last character of the file names:

```
dev      kbytes   used     free
disk0    5044     4124     920      82%
disk1    4984     4422     564      89%
disk1    4984     4422     564      89%
```

or, much more quickly, in the clear on the standard error:

```
df secret news 2>unclassified

dev      kbytes   used     free     use
cannot open /dev/secret
cannot open /dev/news
```

5 One example: create a collection of files named `A`, `B`, `C`, ... each containing one letter, `a`, `b`, `c`, ... A high process opens files to spell out a message and does an *exec* with no arguments. The resulting low process reads from the open file descriptors to receive the message at several hundred bits per second. The channel can be throttled by refusing to

of covert channels should be detectable from audit records.

A mole could certainly use such covert channels to smuggle out precious small secrets to unauthorized users; however an unauthorized user could not exploit them unaided, except by planting a Trojan horse. We supply a special featureless shell to holders of the most powerful licenses to help keep them away from horses. We have also designed audit tools along familiar lines to monitor the stability and safety of security settings.

We undertook this project because we believe it is desirable to try other models than those implied by thoroughgoing adherence to the Orange Book. In particular we suspect that a faithful Orange-Book UNIX system would sacrifice much of the system's productive flavor, with security barriers surprising users at every turn. Dynamic labels should help alleviate the surprises. Moreover, faithful Orange-Book security may be inappropriate in applications where security breaches do not entail risks as final as military defeat. (Commercial users, for example, may recoup damages in court.) In such a setting security priorities are more likely to concern keeping outsiders out, preventing inadvertent leaks by insiders, limiting the chance for mistakes by superusers, frustrating attempts to plant Trojan horses, and reducing the vulnerability of the overall system to a single disaffected superuser—all while maintaining high productivity. Procrustean solutions to curtail covert channels are not so critical.

reduce the label across an *exec* with too many open files.

Multiprocessor UNIX: Separate Processing of I/O

A.J. van de Goor,
Delft University of Technology,
Department of Electrical Engineering,
Mekelweg 4,
P.O. Box 5031,
2600 GA Delft,
The Netherlands.
vdgoor@dutesta.UUCP

A. Moolenaar,
Oce Nederland B.V.,
St. Urbanusweg 126,
P.O. Box 101,
5900 MA Venlo,
The Netherlands.
mool@oce.nl.UUCP

J.M. Mulder,
Delft University of Technology,
hansm@dutesta.UUCP

ABSTRACT

Making UNIX suitable for a multiprocessor system is a logical step because of the wide acceptance of UNIX and the decreasing cost of hardware. The multiprocessor adaptation, however, is not trivial because of some of the assumptions the UNIX kernel is based on. This paper illustrates, on a high level, the performance considerations which guided the design of a UNIX multiprocessor, and it describes specifically the modifications required to implement the I/O kernel layers on dedicated I/O processors. This implementation was based on the concepts of horizontal and vertical data sharing.

1. Introduction

During the last few years the price of microprocessor components has dropped rapidly, while the demand for computer performance has continued to grow. Additionally, the acceptance of UNIX, as an operating-system standard, also increased. A logical conclusion of these observations is that there is a potential need for low cost UNIX multiprocessor systems.

In 1984 a project was started, at Delft University of Technology, with a complete UNIX multiprocessor computer as goal. Both hardware design and UNIX modifications were part of the project.

To be able to run UNIX in a multiprocessor environment the processing load has to be distributed over several processors. Our MP UNIX achieves this through two types of adaptations:

1. UNIX was adapted to be able to run several processes in parallel.
2. The low level I/O functions of UNIX were transferred to I/O processors (IOPs) to relieve the other processors from handling I/O interrupts and buffering.

This article mainly discusses the second type of adaptation. Only the second section touches on the performance aspects of distributing processes over multiple processors; implementation details can be found in [Ja86]. More details on the transfer of I/O to separate processors can be found in [Mo85].

One of the objectives was to adapt UNIX to the multiprocessor environment while making as few modifications as possible. This was necessary because only a limited amount of time (ten months) was available.

Three types of processors are used in the system: the kernel processor or KP, the user processors or UPs, and the I/O processors or IOPs. The kernel processor is physically identical to a UP, but is the only processor which runs kernel code. When the kernel load is low, KP can also run user code. KP delegates UNIX tasks to the user processors and delegates I/O tasks to the I/O processors. On the I/O processors the tasks are executed by a small, UNIX-like, kernel. A remote calling mechanism implements the communication between both kernel and I/O, and kernel and user processors.

A method based on horizontal and vertical data sharing guided the decisions which I/O tasks needed to be transferred from the KP to one of the IOPs. This method is based on [Pa71] and is described in [Mo85]. Despite the complexity of the UNIX kernel, the method helped to keep the number of modifications to a minimum.

Section 2 illustrates some of the performance aspects which were taken into account. Section 3 describes the I/O structure of UNIX and applies the concepts of horizontal and vertical data sharing to determine the appropriate transfer level.

2. Performance considerations

When designing an MP system, it is essential to realize where the weak performance links are to take the appropriate steps for resolving these bottlenecks. The performance considerations presented in this section do not involve complicated models and are not necessarily accurate. They are intended as an illustration. Accurate and more realistic models require advanced queuing models, with parameters obtained through measurements on UNIX systems with realistic workloads.

This section presents some performance considerations when moving from a single processor system via a multi user-processor system to a more complex multiprocessor system with additional I/O processors. The last subsection assesses some of the simplifications and assumptions applied in the first three subsections.

2.1. A single processor system

The total time a set of n processes spends in a single processor system is simply $t_{tot}=t_u+t_k+t_i$, or, in words, the total of user, kernel, and idle time. If sufficient processes are available, then t_i approaches zero and the probability of finding the processor in user mode is $p_u=\frac{t_u}{t_u+t_k}$, and in kernel mode is $p_k=\frac{t_k}{t_u+t_k}$. Examples of p_u and p_k, for a VAX 750 running 4.3 BSD, are 0.9 and 0.1 for a mix of background, interactive, and system jobs on a lightly loaded system, and 0.6 and 0.4 when some kernel (but not I/O) intensive programs are active (e.g. socket-intensive programs like the game hunt).

The time to run a specific job j is $t_j=u_j+k_j+i_j+w_j$, or, in words, the user and kernel time, the time waiting for I/O devices and the time spend waiting in the processor or I/O queues. t_j reduces on a lightly loaded system to $u_j+k_j+i_j$, but, as can be observed quite easily, increases in a non-linear fashion when the load rises.

Given the modification-complexity constraints mentioned in the introduction, the most obvious way of increasing the system throughput is by implementing the kernel on a dedicated processor (KP = kernel processor), and run the user code on a set of user processors (UPs).

2.2. Multiple user processors and a single kernel processor

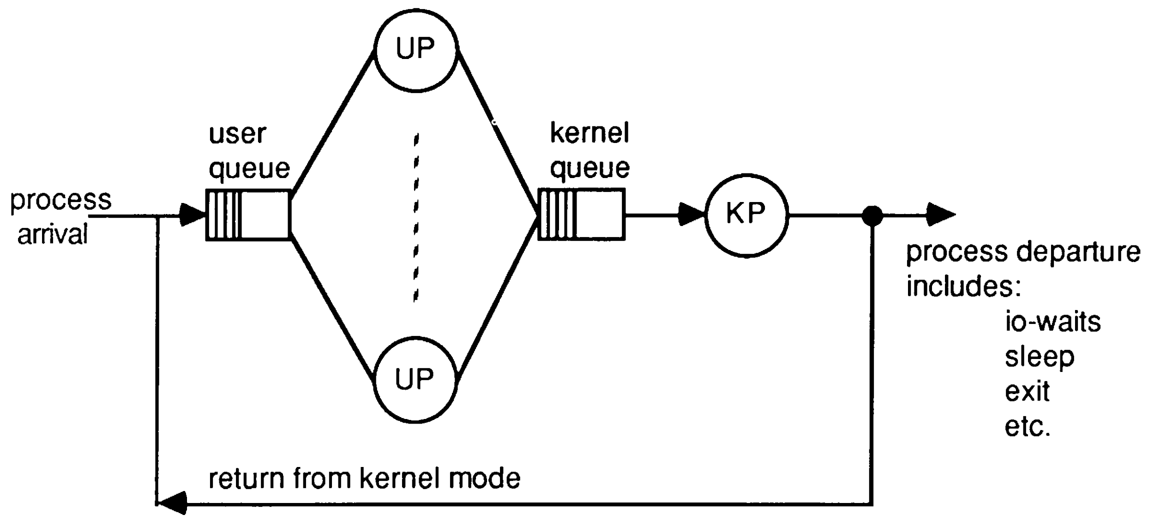


Figure 1. Multiple user processors and a single kernel processor.

The first approach to a multiprocessor is illustrated in Figure 1. To execute a system call the UP calls the KP remotely by placing its request in the KP queue. The UP is free to continue with another user process; after completion of the system call the suspended user process is, again, placed in the UP queue. The exact mechanism, which replaces the system call and return, is not important for the following performance analysis.

Figure 1 already foreshadows that the bottleneck is to be found in the kernel processor. It is useful, however, to estimate how many user processors and processes this system can support, while maintaining sufficient single job throughput and response time.

The system throughput (n jobs) $\frac{n}{t_{tot}}$ is $\frac{n}{t_k+t_i}$ jobs/second, because only observation of the kernel processor is required to find the total execution time. If sufficient user processors and processes are available then $t_{tot}=t_k$ and the kernel processor is fully utilized. If the kernel is not fully utilized, the throughput becomes $\frac{n}{t_{tot}}=n\frac{\rho}{t_k}$ jobs/second, where ρ is the utilization of KP ($0 \le \rho \le 1$).

The single-job execution time $t_j=u_j+k_j+i_j+w_{u_j}+w_{k_j}+o_{rc_j}$, or, in words, the user, kernel, io-wait, user-wait, and kernel-wait time, and the overhead introduced by remote calls. The minimum single-job execution time is $u_j+k_j+i_j+o_{rc_j}$, which is longer than the time to execute the same job on a single processor. The advantage of using an MP system for the single-job throughput is the reduced waiting time, or $w_{u_j}+w_{k_j}+o_{rc_j}<w_j$.

The problem arises, however, when optimizing both system throughput and single-job throughput. Elementary queueing theory shows that when both the arrival and the service time distribution are poisson, the waiting time for the kernel processor $w_{k_j}=\frac{\rho}{\bar{k}(1-\rho)}$, where ρ is the utilization of the kernel processor, and $\bar{k}$ is the average time a job spends executing in the kernel. This queueing model (M/M/1, e.g. see Kleinrock Vol I and II) does not completely apply, it is better to leave the exact model in the middle and approximate the utilization by $w_k=\rho\frac{c}{1-\rho}$, where c is a constant, which depends on the distribution of the kernel-execution times (M/G/1 queueing model).

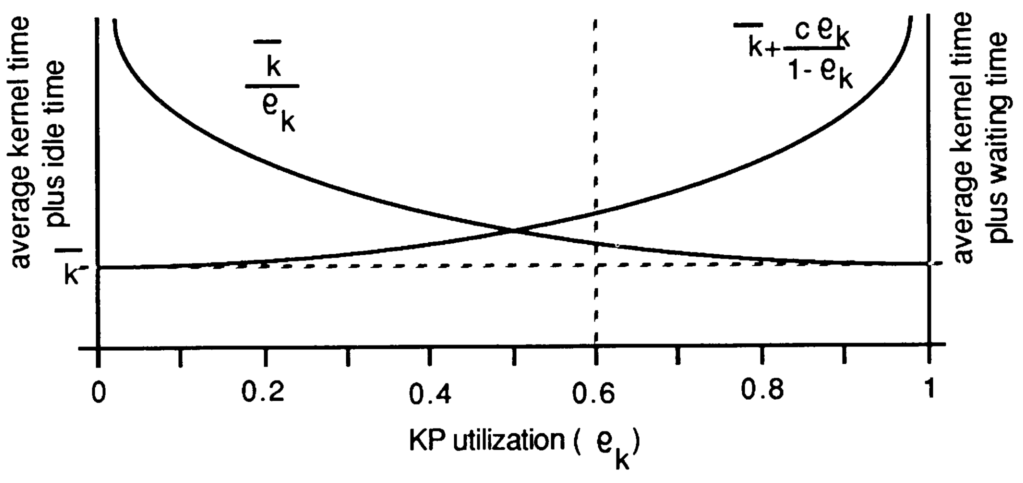


Figure 2. Tradeoff between system and single-job throughput.

The tradeoff between system throughput and single-job throughput, shown in Figure 2, is contradictory. When the utilization increases the system throughput in creases, but the single-job throughput reduces because the queue waiting times become very long. A rule of thumb, used, generally, in interleaved memory design, is a ρ of 0.6 for the utilization of the shared bus. Applying the same rule to the MP design yields an KP utilization of 0.6, which implies that KP is active 60% of the time, or that the probability of finding KP idle is 0.4. By means of p_k and ρ, one can calculate the number of fully utilized user processors $N = \frac{\rho}{p_k}$ (p_k is defined in section 2.1). Fully utilized user processor, however, yield high user-queue waiting times (w_{u_i}). Under the assumption that process migration is penalty free[†] , the probability that an arriving user process finds at least one free user processor is $p_{nowait} = 1 - \rho_u^N$, where ρ_u is the average individual utilization of the UPs. Requiring that p_{nowait} equals $1 - \rho_k$ yields $\rho_k = 0.6 = \rho_u^N$. Introducing ρ_u in the calculation of N yields $N = \frac{\rho_k}{\rho_u p_k}$, which combined with calculation of ρ_u yields

$$p_k = \frac{\rho_k}{N\rho_u} = \frac{\rho_k}{N\rho_k^{1/N}} = \frac{\rho_k^{\frac{N-1}{N}}}{N}.$$

[†] When processes migrate without penalty from processor to processor, then it is not necessary to schedule a process always on the same processor. This assumption is, generally, not valid, because moving the complete state of a process is expensive.

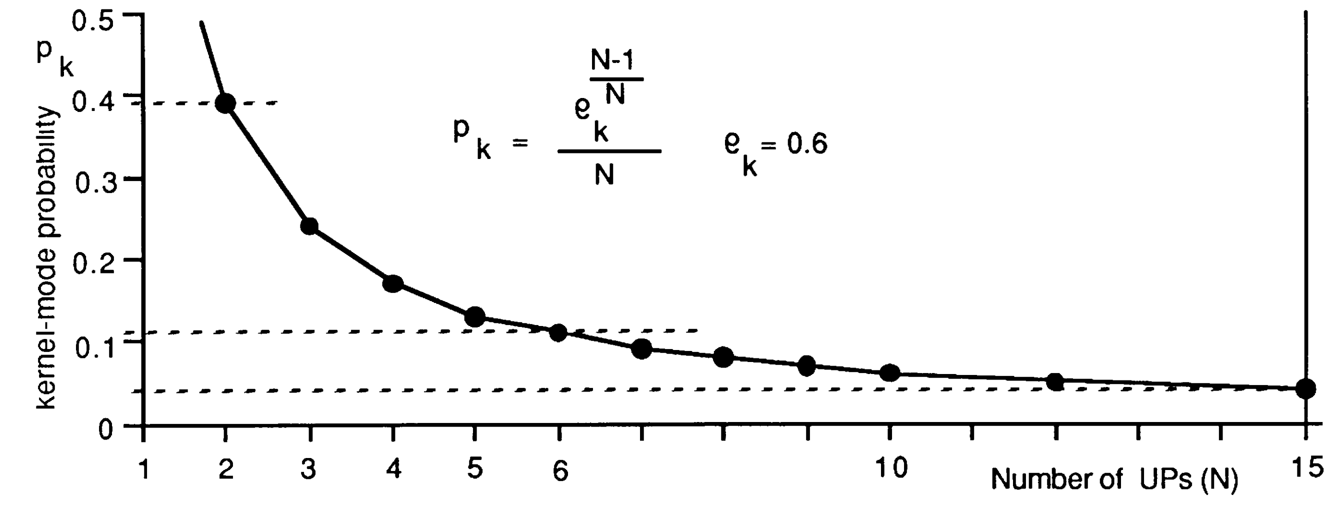


Figure 3. p_k as function of the number of UPs (N).

Figure 3 shows the maximum allowable p_k as function of N, when ρ_k=0.6. Referring to the examples of p_k mentioned in section 2.1, 6 processors yield 10% and 2 processors yield 40%.

Obviously, the kernel time needs to be reduced to allow both a higher system throughput and to reduce waiting times in the kernel queue. The most obvious way of reducing kernel time, which also requires the least amount of kernel modifications, is the transfer of I/O specific functions from KP to dedicated I/O processors (IOPs).

2.3. Addition of I/O processors

A logical step in improving the system throughput, is reducing the kernel load by offloading the I/O code to dedicated I/O processors (IOPs). Figure 4 shows the resulting model; instead of included in the kernel node of the model of section 2.2, the I/O bottleneck appears explicitly with a set of processing nodes (IOPS) each with its own queue. Addition of I/O processors, however, requires an investment which need to be justified. In this section some conditions are layed out for this justification.

In the previous section the silent assumption was made that the resource which limits the performance of the kernel is KP itself. However, creating a system with, for example, 6 user processors automatically yields a significant increase in I/O when compared with a single processor system. To be more exact, the average I/O requirements increase with a factor $\frac{N\rho_u}{p_u}$ over the, fully utilized, single processor requirements (6 times when N=6, ρ_u=0.9, and p_u=0.9). The average I/O system throughput needs to be increased by at least that factor to keep the KP busy. This section assumes that the total bandwidth of all I/O devices is more than sufficient to process the increased demand. Note, however, that, in a more realistic MP performance model, the I/O bandwidth in general and the disk/network bandwidth specifically appear as performance limitations. Using very large caches (≥4 Mbyte) for local and remote disks may be necessary to resolve this I/O bottleneck [OD85].

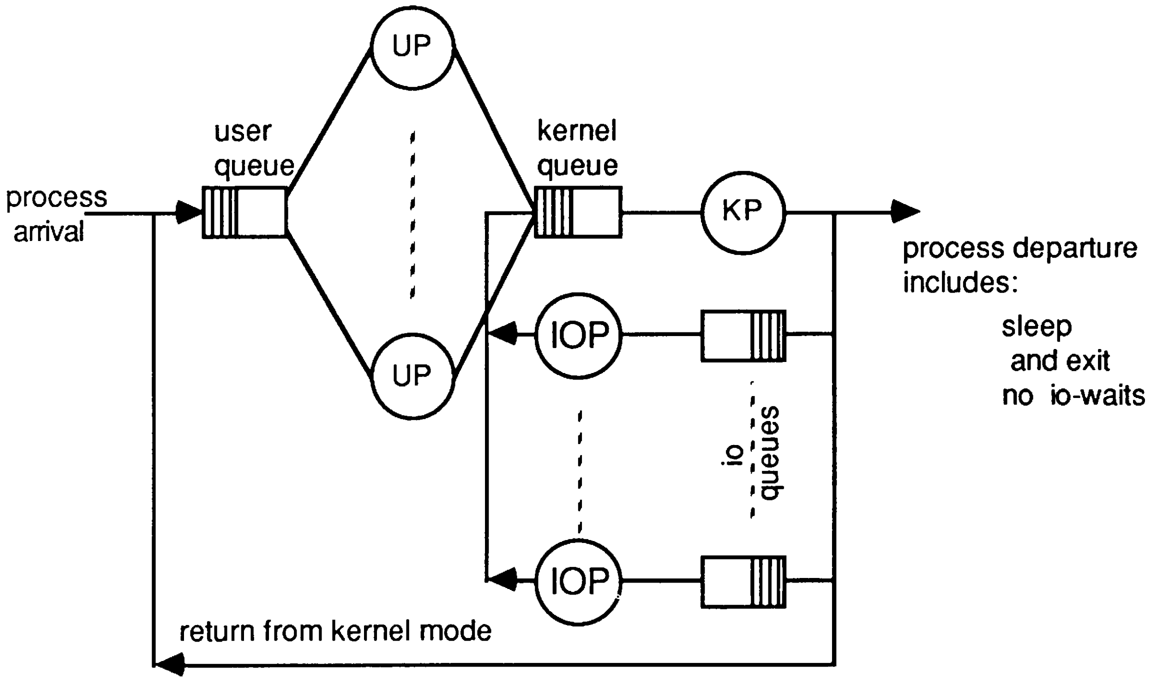


Figure 4. MP organization with one KP and multiple UPs and IOPs.

By means of the simple model explained in section 2.2, and shown in Figure 3, the number of user processors can be calculated for the reduced kernel activity, p_k. For example, when removing I/O processing from the KP reduces p_k to 4%, the maximum allowable number of user processors increases to 15. The average I/O bandwidth — when compared with, fully utilized, single processor system — increases also with a factor 15 (ρ_u=0.9 and p_u=0.9).

Similarly to the increase of single-job execution time when distributing the user and kernel code over different processors, this execution time increases again when the kernel is distributed over the KP and IOPs because of the overhead incurred when calling remotely. The reduced load on the KP and the possibility to increase the number of UPs, however, compensate the introduced overhead by reducing the both user and kernel waiting times.

2.4. Conclusion

The main drawback of the performance model presented in the previous sections is the assumption that processing power is the bottleneck. This assumption is valid when few UPs are available, which communicate only with the KP. Many UPs together with substantial interprocessor communication will undoubtly clog the bus and memory system before the system runs out of processors. Keep this in mind when interpreting these simple calculations; they are meant to illustrate the potential performance of very simple and low-cost UNIX MP systems.

Relieving the KP from its I/O functions is potentially a powerful way of boosting the system performance when the following conditions are met.

1. p_k should reduce significantly. Note that calculating the kernel-execution time now includes the remote-call overhead incurred for every transferred I/O call.
2. The potential increase of the system throughput should not move the bottleneck to the bus and memory system or the I/O devices.
3. As long as the kernel processor is not the system bottleneck, introducing I/O processors will improve neither average system nor average single-job throughput.

The following section assumes that these conditions are met, and describes the transfer of I/O functions to dedicated I/O processors.

3. UNIX I/O transfer

Sections 2.3 and 2.4 presented the motivation for separating I/O from the kernel processor, and outlined some conditions to be satisfied before separate I/O processors become useful. This section describes the process of transferring the I/O system to separate processors. First, the section introduces the UNIX I/O system, and, second, determines the level of transfer for the two main I/O sub systems (block and character I/O).

The section refers regularly to the concepts of vertical and horizontal data sharing [Mo85].

Vertical data sharing.

When I/O functions, which are transferred to an IOP, share data with functions which remain on the KP.

Horizontal data sharing.

When I/O functions, which are transferred to different IOPS, share data.

This section starts with an overview of the I/O subsystem of UNIX, after which the transfer level of the two main I/O parts (block I/O and character I/O) are discussed.

3.1. UNIX I/O layers

There are many ways to group the I/O functions of the UNIX kernel. The one presented employs a hierarchical model, much like the one used by Peppinck [Pe84]. In this model the I/O is divided into four layers. A function in one layer *uses* functions in the same layer and functions of lower layers. The *uses* relation is defined by the following rule‡:

Function A *uses* function B if A invokes B and depends upon the results of that invocation.

As an example, suppose there are three layers. The functions in the highest layer may *use* all three layers; the functions in the middle layer may *use* the lower two layers, but will not *use* functions in the highest layer. In general, the *uses* relation merely means that one function invokes another function. But on some occasions this can yield problems in building a hierarchical model. Suppose function A invokes function B, which invokes function C, which invokes function A again. It is not possible to determine a hierarchy from these invocations. But if one knows that function C does not depend on the results of function A, we can say that function A lies above function C in the hierarchy.

The separation between I/O layers is not always as clear as it should be. This is due to the fact that the UNIX kernel was not written in a hierarchical way. However, when considering I/O transfer, the following four layers will suffice, see Figure 5 (for more details [Pe84] page 111):

1. the system call layer
2. the i-node* layer
3 the buffer cache layer
4 the device driver layer

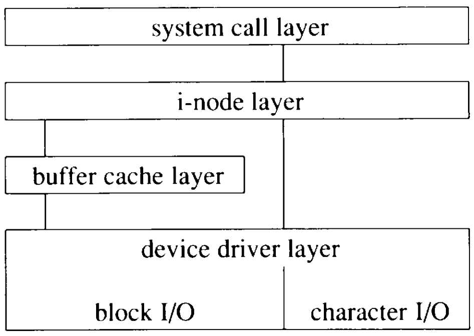


Figure 5. UNIX I/O layers.

The system call and i-node layers will be called the *higher* layers, by virtue of the fact that these two layers implement the UNIX file system. The *lower* layers, buffer cache and device driver, merely transport data

‡ This definition is based on the original definition of the uses relation by Parnas [Pa78].

* An i-node is the internal representation of a file. It is a data structure that contains all attributes of a file (length, creation date, ownership, pointers to the contents of the file, etc.), except for the name.

and do not project a structure on this data.

In the *lower* two layers a distinction is made between block I/O and character I/O. These two parts are kept completely separate. Block I/O is done with blocks of data with a fixed length (e.g. for disk transfers) and character I/O with sequences of bytes of any length (e.g. for communication with terminals). The buffer cache is used for block I/O only. Buffering for character I/O is carried out at the device driver layer. The global definitions for the I/O layers are listed below:

system call
: The functions in this layer translate the I/O service requests (system calls) to operations on the actual files in the file system. To allow for this, a data structure is maintained for each process, containing information about all files opened by that process.

i-node The functions in this layer execute the operations on files (which in fact are operations on i-nodes), which result in operations (I/O requests) on devices. It is figured out which device is associated with the file. Data controlled in this layer is the i-node cache† .

buffer cache
: In this layer the buffer cache is controlled. The buffer cache is a software cache mechanism which is used to speed up the access to block I/O devices. It contains the most recently used data of the block I/O devices.

device driver
: This layer contains all the device drivers. A device driver forms the interface between the device independent part of the kernel and a device. There is one device driver for each type of device.

3.2. High transfer level

To relieve the master, as many I/O functions as possible should be transferred to IOPs. But the device on which the I/O will be performed is not known until in the i-node layer (the system call layer doesn't know about devices, it merely uses files). So when the i-node layer is transferred to IOPs, the master does not known which IOP should handle the I/O. Two methods to solve this problem will be considered.

The *first* method is to add a file processor. This processor executes the i-node layer and possibly the system call layer. It calls the IOPs to execute the lower layers. This new architecture is illustrated in Figure 6.

† The i-node cache is a software mechanism that speeds up the access to the file system by storing the i-nodes of all open files (i.e. files which are opened but not closed yet).

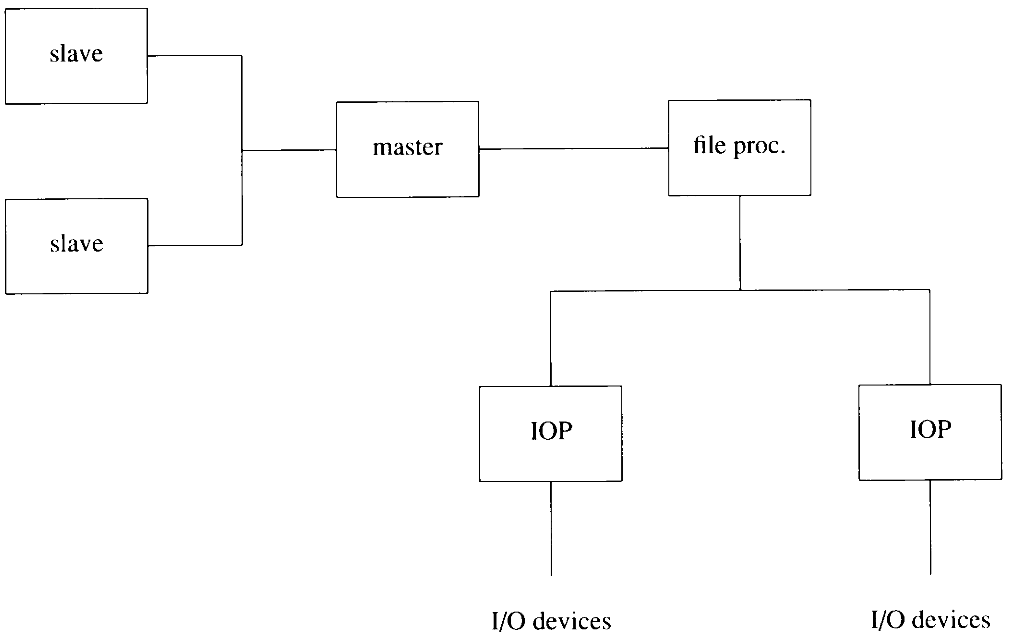


Figure 6. Multiprocessor configuration with file processor.

A significant disadvantage of this system is that the file processor could become another bottleneck in large systems. In addition, the system gets more complicated and the path from a slave to an IOP gets longer, which introduces extra delays. Because of these disadvantages and because simplicity was one of the goals, the file processor was not used.

The *second* method is to choose an IOP at random, let this IOP handle the I/O down to the i-node layer and then, depending on whether it controls the device or not, continue with the I/O or hand it over to the IOP that does control the device. There are some disadvantages to this method:

1. The i-node layer will be executed by all IOPs. Therefore, the data used in this layer, the (software) i-node cache, will be accessed by all IOPs and must be placed in global memory. And because the IOPs, unlike UPs, do not have a (hardware) cache, this will increase the bus load.
2. The I/O will first be handled by one IOP and then possibly handed over to another IOP, which entails extra delays.
3. User address space is accessed in the i-node layer. It is quite difficult to access user space from IOPs. It is certain to cause complexity and may even introduce extra delays.

In light of these reasons the i-node layer was not transferred. The system call layer will not be transferred either, because it is highly illogical and unpractical to transfer the system call layer when the i-node layer is not transferred. The question of whether the lower layers should be transferred or not will be handled for block and character I/O separately in the next two sections.

3.3. Block I/O transfer level

There are two I/O layers involved in block I/O: the buffer cache layer and the device driver layer. The device driver layer will certainly be transferred, as otherwise there would be no transfer at all. In order to decide whether the buffer cache layer will be transferred, the horizontal and vertical data sharing must be considered. In Figure 7 the main data structures for block I/O are depicted.

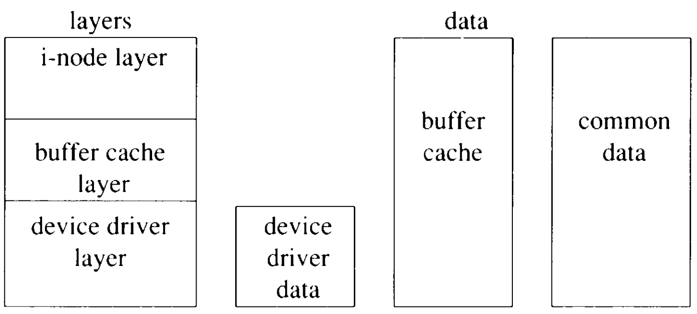


Figure 7. Block I/O trañsfer levels and data.

Vertical data sharing

As can be seen in Figure 7, the buffer cache is accessed in both the i-node layer and the device driver layer. This implies that the vertical data sharing of the buffer cache cannot be avoided, because the device driver layer will be transferred and the i-node layer will not.

Because of this vertical data sharing, the buffer cache should be protected via mutual exclusion. Fortunately, there is already some form of mutual exclusion for the buffer cache. The functions in the i-node layer must first request a buffer before they can use it, and they will release the buffer afterwards. This mechanism can be utilized for mutual exclusion of the buffer cache. There is yet another mechanism which can be used. Between the buffer cache layer and the device driver layer, buffers are "handed over". The device drivers are invoked to perform I/O on a buffer and when the I/O is complete this is signaled back to the invoking function. This is a very primitive mutual exclusion mechanism.

Because of the two existing mutual exclusion mechanisms, the transfer level could be at the buffer cache layer or at the device driver layer. However, the mutual exclusion mechanism between the i-node layer and the buffer cache layer is more advanced. When vertical data sharing is considered, the buffer cache level is somewhat simpler to implement than the device driver level.

Horizontal data sharing

Another problem occurring when transferring the buffer cache layer is horizontal data sharing. The device driver data should not cause problems, because this data is device dependent, meaning that each IOP can have its own device driver data. The buffer cache could be a source of problems. The buffer cache is a pool of buffers, which are not associated with a specific device. As mentioned in the section on data sharing, this pool could be split into several pools, one for each IOP.

When one buffer cache is used for the whole system, buffers are not statically associated with any device. When one buffer cache is used for each IOP, buffers will always be associated with a group of devices. To what extent the performance is affected by this is hard to tell. It depends on the sequence of accesses to the block I/O devices. It is even possible for performance to be improved, because each IOP could have its buffer cache in local memory, thereby decreasing the bus load. Whatever the performance change, it can always be compensated by modifying the size of the buffer cache (at the cost of some memory, which is assumed to be of minor importance). Therefore the performance aspect will not be taken into consideration when deciding at which level block I/O should be transferred.

When only the device driver layer is transferred, there are two possibilities:

1. Using one buffer cache for each IOP. The buffer cache layer will then have to be modified, because not one but several buffer caches exist, one of which must be selected.
2. Using one buffer cache. The buffer cache layer will then not have to be modified to accommodate the multiple buffer caches, but the access from the device drivers to the buffer cache must be modified at some points for mutual exclusion.

Although it is difficult to choose between these two possibilities, it is clear that modifications cannot be avoided.

When the buffer cache layer is transferred there should be one buffer cache for each IOP, because having one common, system wide, buffer cache entails quite a lot of modifications. The number of modifications involved in multiple instances of the buffer cache (one per IOP) is quite small. This stems from the fact that the higher layers don't know where the buffers of the buffer cache are placed and consequently have to ask this from the functions in the buffer cache layer by requesting a buffer.

When the decision about transfer level is based solely on horizontal data sharing, the buffer cache level is the best choice.

Conclusion

Transferring block I/O at the buffer cache level is a good choice, as it serves both vertical and horizontal data sharing the best.

3.4. Character I/O transfer level

The device driver layer for character I/O is quite large. This allows for a split up into several other layers. So one could consider to transfer only a part of the device driver layer. First an important decision has to be made: from which processors can the tty structures‡ be accessed? This decision is important because the tty structures are used by almost all functions in the character I/O layer. There are three possibilities:

1. If only the master is allowed to access the tty structures, only the functions which don't access the tty structures can be transferred, which are only a few.
2. If both the master and the IOPs are allowed to access the tty structures there is vertical data sharing. Therefore mutual exclusion has to be added, causing a lot of modifications.
3. If only the IOPs are allowed to access the tty structures, then the character I/O layer must be transferred completely. This causes horizontal data sharing. But because the tty structures are each related to a specific device, this horizontal data sharing can easily be avoided.

Clearly the third possibility is the best. Transferring the tty structures implies that all character I/O code must be transferred to IOPs. At first glance this might seem to invoke a great deal of work (about 2400 lines of source code are involved), but it actually doesn't. The main advantage is that the module, which is formed by the character I/O code, is not split up by the transfer. As a result it does not matter how character I/O works internally. Only the interfaces between the character I/O part and the rest of the kernel have to be examined, of which there are fortunately only a few.

4. Conclusion

With simple hardware means and with little software modification UNIX can run on a multiprocessor system. This MP organization is modeled strongly after the internal structure of UNIX, by allowing multiple processors to run user processes, but to allow only one single processor to run the kernel.

When performance is mainly limited by the processing power of the kernel processor, the transfer of I/O functions to separate I/O processors improves the performance, creates a potentially device-independent kernel, and improves the functional separation within the UNIX kernel. Note, however, that both I/O devices and bus/memory system are potential bottlenecks which are much harder to relieve than limited kernel processor power.

It is possible to transfer some of the I/O functions of the UNIX kernel to I/O processors and make only a small number of modifications. To do this, the transfer level has to be chosen in such a way that data sharing between processors is avoided. The lower I/O layers can be transferred easily with only a small number of modifications, because the data which is used in these layers is related to a device, or can be made to be related to a (group of) device(s), so that this data is used by only one IOP. The higher I/O layers cannot be transferred easily, because the data which is used in these layers is not related to a particular device, so this data would be shared between several processors.

The transfer of the character I/O part is easy to accomplish because there is hardly any data sharing between the character I/O part and the rest of the kernel. One could say that the character I/O part is one module, which contains all the design decisions for character I/O. Once again, this illustrates the advantage of the modularization as proposed by Parnas [Pa71].

The transfer, as presented here, has only been implemented partially. The IOP kernel was designed and tested. The communication mechanism was designed but not tested. All modifications which have to be applied to the kernel when the block I/O part is transferred have been identified [Mo85], so no problems should arise during implementation. The modifications which are necessary for the transfer of the character I/O part are not listed, but all areas where problems could arise have been examined and solutions to these problems have been given.

‡ A tty structure contains all information about a terminal: pointers to the input and output buffers, status, associated processes, etc.

5. References

[An85] J.K. Annot and M.D. Janssens, *Multiprocessor UNIX,* Master's Thesis, Department of Electrical Engineering, Delft University of Technology, June 1985.

[Go85] A.J. van de Goor and R.J. Bril, *On Cache Schemes for DUMP1,* to be published.

[Ja86] M.D. Janssens, J.K. Annot, and A.J. van de Goor, *Adapting UNIX for a multiprocessor Environment,* ACM Computing Practices. September 1986. Volume 29. Number 9.

[Mo85] A. Moolenaar, *Transferring UNIX I/O to I/O Processors,* Master's Thesis, Delft University of Technology, Department of Electrical Engineering, July 1985.

[Pa71] David L. Parnas, *On the criteria to be used in decomposing systems into modules,* Department of Computer Science, Carnegie-Mellon University, Pittsburgh, August 1971 (also published in Comm. ACM, vol. 15 nr 12, December 1972).

[Pa78] David L. Parnas, On a "Buzzword": Hierarchical Structure, in: *Programming methodology,* A collection of articles by members of IFIP WG 2.3, edited by David Gories, Springer-Verlag, New York, 1978, pp 335-342; also in: *Proceedings of IFIP Congress 74,* 1974, North- Holland Publ. Co.

[Pe84] W.R. Peppinck, *Modeling the UNIX kernel,* Master's Thesis, Department of Electrical Engineering, Delft University of Technology, June 1984.

[OD85] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, J. Thompson *A Trace-Driven Analysis of the UNIX 4.2 BSD File System* ACM SIGOS. Operating Systems Review. Vol. 19, no. 5, p. 15-24.

Word Manipulation in Online Catalog Searching: Using the UNIX System for Library Experiments

Michael Lesk

Department of Computer Science
University College London
Gower St
London WC1E 6BT

Bellcore
435 South St
Morristown, NJ 07960

ABSTRACT

Online public access catalogs are often plagued with very short queries and very short document descriptions. As a result performance may be poor and the users are dissatisfied. To improve recall, in particular to deal with query terms not found in the collection, a machine-readable dictionary can be used to identify related terms by overlap of defining words. To improve precision, phrases can be retrieved and the user asked to pick the appropriate ones. A demonstration system is running on 72,000 records from the British Library Eighteenth Century Short Title Catalog.

A UNIX system is a good way to implement this software, because of its advantages of easy programming, availability on small machines, and advanced data base routines.

Introduction.

"Libraries for books will have ceased to exist in the more advanced countries except for a few which will be preserved at museums" – so wrote Arthur Samuel in 1964, predicting the world of 1984 for the *New Scientist*.[1] Although this schedule has slipped, we are finding more and more libraries with online catalogs and many online full text data bases are routinely accessed around the world. Unfortunately sometimes users are not happy with the resulting systems.[2] This paper describes techniques for improving searching in retrieval systems, particularly OPACs (Online Public Access Catalogs). To summarize, retrieval is suffering because searching is inadequate, being based on very few terms, and because browsing capabilities are inadequate, offering too few retrieved documents and too little chance to decide which are relevant. By using the flexible software environment on UNIX, it is possible to experiment quickly with software to help.

First, let us consider the problems caused by searching too few terms. Queries typed by users to OPACs are often very short; in one experiment an average length of 1.5 words was seen.[3] Users who are accustomed to traditional catalogs, of course, will expect that they can look up only one heading at a time. Simultaneously the document representations, the catalog records, are also very short: a title plus perhaps a few subject headings. The combination of short queries and short representations means that there is only a small chance of overlap. Many relevant documents will not be found, either because the terms used in them are not ones the user thought of, or because the user neglected to enter the terms even after thinking of them. Or, too many documents may be found, due to the large size of library catalogs and the great frequency of some words. Thus automated retrieval suffers, and users who think that an online catalog is a good way to do subject access searching may be frustrated.

[1] A. L. Samuel, *The banishment of paperwork*, 21, pp. 529-530, New Scientist, 27 February 1964.

[2] C. Borgman, "Why are online catalogs hard to use? Lessons learned from information retrieval studies.," *J. Amer. Soc. for Information Science*, vol. 37, no. 6, pp. 387-400, November 1986.

[3] V. J. Geller and M. E. Lesk, "An Online Catalog Offering Menu and Keyword User Interfaces," *Proc. 4th National Online Meeting*, pp. 159-166, New York, April 1983.

Browsing may also be frustrated by some kinds of online catalogs. If the catalog presents only the exact items found, it may be difficult to know what was nearby, or almost found. By contrast it is not possible to look in a printed catalog without seeing the whole page, or to flip through cards without seeing the adjacent cards. An OPAC is more useful if there is some ability to see what was near the sought items, as well as the exact matches to the query.

It would also be useful to have more information about each item. Ideally, there would be a complete gradation of available information from the title to the full book complete with all graphics and illustrations. In reality any addition to the title and subject headings (e.g. the table of contents) would be helpful, but it is not clear when we might get such information.

To return to the problems of short queries, they may be either recall problems (not enough relevant material is found) or precision problems (too much irrelevant material is found).[4] Recall failure normally manifests itself as "no hits" or some such diagnostic indicating that the user's query hasn't matched anything. Precision failure normally represents a huge amount of retrieved material. Efforts have been made to deal with both, since the program can tell which problem exists by counting the number of retrieved documents it is going to display. If that number is zero, it tries to improve recall; if the number is greater than 100, it tries to improve precision.

Traditionally, recall improvement involves a thesaurus.[5] These are made by hand, and involve significant effort to keep up to date.[6] Traditional thesauri, by replacing a variety of synonyms with a single term, permit the detection of overlap between concepts originally phrased in different words. Thesauri also help the user adjust a query by displaying the various possible indexing terms in the vicinity of those used to start with, allowing the user to select additional terms or phrases from the query.

Efforts have been made to construct thesauri automatically from a dictionary, e.g. the work done with Merriam-Webster's 7th New Collegiate at IBM.[7] However, it is not necessary, and possibly not even desirable, to make a traditional printed thesaurus for retrieval purposes. Thesauri have the disadvantage that they have to be consulted, and often the users do not wish to take the time to do that; nor do end users often have the expertise to understand how and why the thesaurus should be employed. Finally, for some queries the forced combination of subjects may impede retrieval (by folding together concepts the user might wish to keep separate), although in a competently made thesaurus this problem is anticipated and generally avoided during the construction of the indexing language described by the thesaurus.

To improve recall without a thesaurus, dictionary definitions in machine-readable form can be used to expand queries into words, which although not synonyms, are often useful. For example, augmenting "tuberculosis" with "consumption" is sensible with older books; and even adding "native" to a query for "kangaroo" turned out to be valuable (because of books with titles such as "A voyage to New South Wales with a description of the country, manners ... of the natives ..." Dictionaries have not been used much for retrieval, although in many ways a dictionary is merely a thesaurus sorted on the alphabetical words rather than their meanings.

This program employs dictionaries to provide vocabulary expansion. There are also various other vocabulary devices, including phrase searching, relevance feedback, and the like. To help with browsing, the system uses a bitmap screen to provide a graded interface from one-line listings, full citations, possibly images of title pages or dust jackets, and complete text where available. It is hoped that this additional information about books will ease the task of the reader, and require fewer books to be scanned by eye, although so far insufficient data is available to perform any experiment.

Demonstration Program.

At present the catalog program runs with a subset of 72,000 book records from ESTC. The ESTC office provided a tape of every other record from the file (for security reasons) and I extracted those items which were held by the British Library (so that any book that was in the file would be relatively easy to find, and in an effort to remove some items of low importance).

[4] G. Salton and M. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, 1983.

[5] *Inspec Thesaurus*, IEE, London, 1985.

[6] S. M. Humphrey, ''File Maintenance of MeSH Headings in MEDLINE,'' *J. Amer. Soc. Inf. Sci.*, vol. 35, no. 1, pp. 34-44, 1984.

[7] M. Chodorow, R. Byrd, and G. Heidorn, ''Extracting Semantic Hierarchies from a Large On-Line Dictionary,'' *Proc. 23rd Annual ACL Meeting*, pp. 299-304, Chicago, 1985.

The program resembles a traditional OPAC, except that it does not prompt for "kind of search". It merely asks you to type words, and then it does a coordinate index search through the entire file, for whatever it can find. Thus, for example, the input *kangaroo* produces the following citation:

```
T: The voyage of Governor Phillip to Botany Bay;; with an account of the
   establishment of the colonies of Port Jackson & Norfolk Island;
   compiled from authentic papers, ... embellished with fifty five
   copper plates. ...
#: Bound in Kangaroo skin
C: London
I: printed for John Stockdale
D: 1789
O: [2],6,[2],viii,[12],x,298,lxxiv,[2]p.,plates; maps; 4
O: The titlepage is engraved
```

It is possible, by using suffixes, to select particular fields. For example, suppose the query word typed is **Bromley**; this finds 37 items, the first few titles of which look like this:

```
A sermon preached at Fitz-Roy chapel, on occasion of the general  Bromley, Robert Anthony / 1789
The state of Bromley College in Kent. / Bromley College; Kent / 1735
A sermon on the nature of subscription to articles of religion;; / Burnaby, Andrew, 1732-1812 / 1774
The mayor, bayliffs, and commonalty : the city of Coventry, / Coventry; Warwickshire / 1721
An accurate description of Bromley,; in Kent, ornamented with views of / Wilson, Thomas, Bookseller, Bromley /
```

Note that these are a mixture of items about the place, those written by someone named Bromley, and those published in Bromley. By specifying **bromley/A** one requests Bromley in the author field, and gets only 16 items of the 37, some of which are:

```
An address from a clergyman to his parishioners, by William Bromley / Cadogan, William Bromley, 1751-1797 / 178
A sermon, preached at Fitzroy Chapel, in London: on Thursday the 29th. / Bromley, Robert Anthony / 1798
A philosophical and critical history of the fine arts,; painting,  Bromley, Robert Anthony / 1793-95
The way to the Sabbath of rest:; or the soul's progress in the work of / Bromley, Thomas / 1744
```

Alternatively, one can use multiple query words, which are searched by coordinate indexing (as many words as possible) to produce a ranked output. For example, to find items on Dick Whittington's cat, one may search for **Whittington cat**, which yields the following titles:

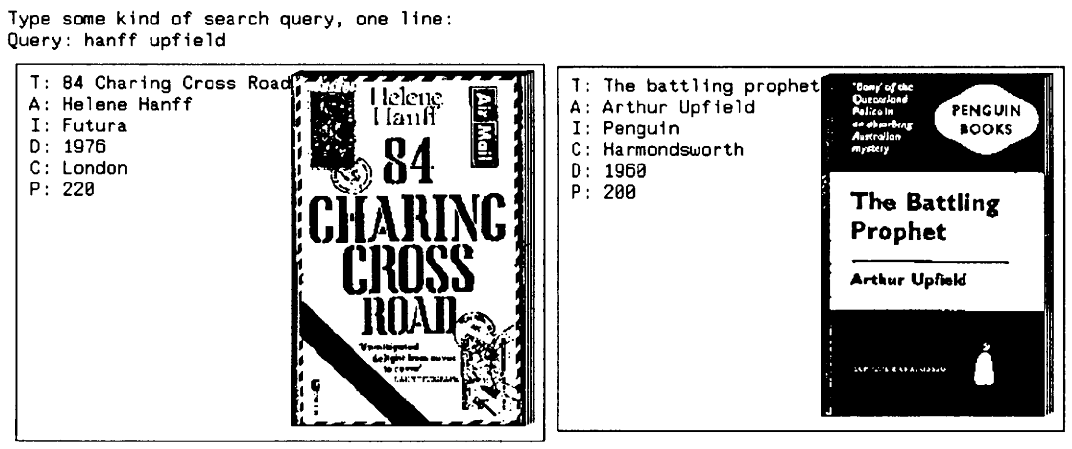
```
An old ballad of Whittington and his cat. /  / [1750?]
The famous history of Dick Whittington and his cat: shewing, how from /  / 1788
The history of Sir Richard Whittington, thrice Lord Mayor of London. /  / [1790?]
The history of Sir Richard Whittington, thrice Lord Mayor of London. /  / [1770?]
The advantages of a good name, and godly end.; A sermon, occasioned by / Wright, Paul / 1773
The lord's lamentation; or, the Whittington defeat /  / 1747?]
An act for rendering effectual an act, made in the seventeenth year of / Great Britain; ..
Whittington reviv'd or the city in triumph: on Alderman Parson's being /  / 1740]
A sermon preached at Lancaster,; / Ravald, Robert, Rector of Whittington / 1766
Hell upon earth: or the most pleasant and delectable history of / Tuus inimicus / 1703
A comical and diverting dialogue, between a Spanish devil of the /  / 1729?]
Select poems from the works of Thomas Gray :; viz. An ode on spring. / Gray, Thomas / 1795
A brief history of the Kings of England, particularly those of the / Weldon, Sir, Anthony / 1755
The Jacobite cat and parson. /  / [1785?]
The Kit-Cats. A poem. / Blackmore, Sir, Richard, d. 1729 / 1708
The eagle and the robin.; An apologue. Translated from the original of / H. G. / 1709
The courtship of the cats. /  / [1785?]
The fable of the shepherd and his dog, in answer to the fable of the /  / 1712
Hush cat from under the table. To which are added II. Tweed-side. III. /  / 1787
A sorrowful ditty; or, the lady's lamentation for the death of her /  / 1748
The wandering young gent[l]ewoman; or, cat-skin's garland. ... /  / 1800?]
When the cat's away, the mice may play. A fable, humbly inscrib'd to /  / [1712]
The wonder of wonders: or, a true and perfect narrative of a woman /  / 1726
Spanish amusements: or, the adventures of that celebrated courtezan / Castillo Solborzano,
Recueil pr'ecieux de la Maconnerie Adonhiramite.; Contenant les / Guillemain de Saint-Victor,
The pole-cat;; or, Charles Jennings, the renegado school-master, of / Shebbeare, John, 1709
```

Note that the first two items contain both ''Whittington'' and ''cat;'' the remaining items have only one of these words. Term coordination can also be used as a way to disambiguate. If one searches **lamb**, for example, one finds a mixture of cookbooks, sermons on the ''lamb of God'', locations specified relative to a pub named the Lamb, various individuals named Lamb as authors, subjects, and publishers, a song entitled ''Tommy Lamb's cure for a drunken wife'' and other such. But searching for **lamb sheep pasture wool** yields as the first two citations:

```
T: Report from the Committee appointed to consider the several laws now
   in being, for preventing the exportation of live sheep and lambs,
   wool, wool fells, ...
A: Great Britain; Parliament; House of Commons
#: B.S.[vol.38]87
O: Proceedings. 1788-03-05
C: [London]
I: Printed in the year
D: 1788
O: 54p.; 2
O: Issued as a Parliamentary paper

T: Observations on the different breeds of sheep, and the state of sheep
   farming, in the southern districts of Scotland: being the result of a
   tour through these parts, made under the direction of the Society for
   Improvement of British Wool. By Mr. John Naismyth at Hamilton
A: Naismith, John, Writer on Agriculture
C: Edinburgh
I: printed by W. Smellie
D: 1795
O: [4],75,[1]p.; 4
O: Half-title: 'A tour through the sheep pastures in the southern parts
O: of Scotland'
```

Levels of Display.

As shown above, the program has various levels of display that it can show. Titles and citations are clearly the most useful. As a investigation of the use of graphics, the ability to display dust jackets or covers, to show book appearances, was included. Figure 1 shows a sample display (abbreviated for legibility).



```
Type some kind of search query, one line:
Query: hanff upfield

T: 84 Charing Cross Road        T: The battling prophet
A: Helene Hanff                 A: Arthur Upfield
I: Futura                       I: Penguin
D: 1976                         C: Harmondsworth
C: London                       D: 1960
P: 220                          P: 200
```

Figure 1.

Attempting to use this feature for items such as maps, where it might be thought that some indication of appearance would be useful, failed because of the inability to represent enough detail on the screen; see Figure 2.

```
Type some kind of search query, one line:
Query: coxwold counties
```

```
T: Map of British counties
I: Ordnance Survey
D: 1974
```

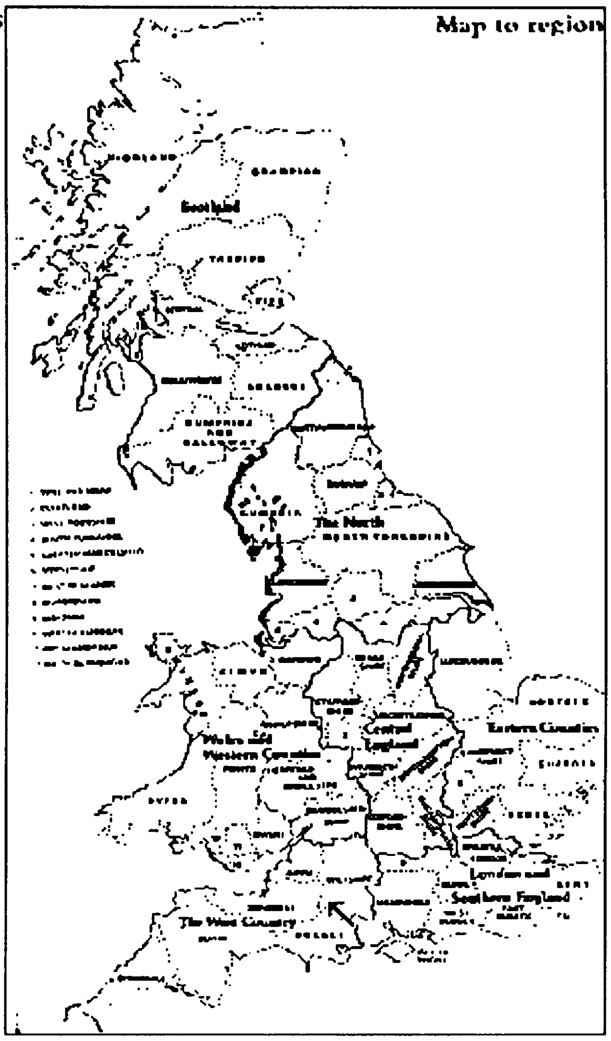
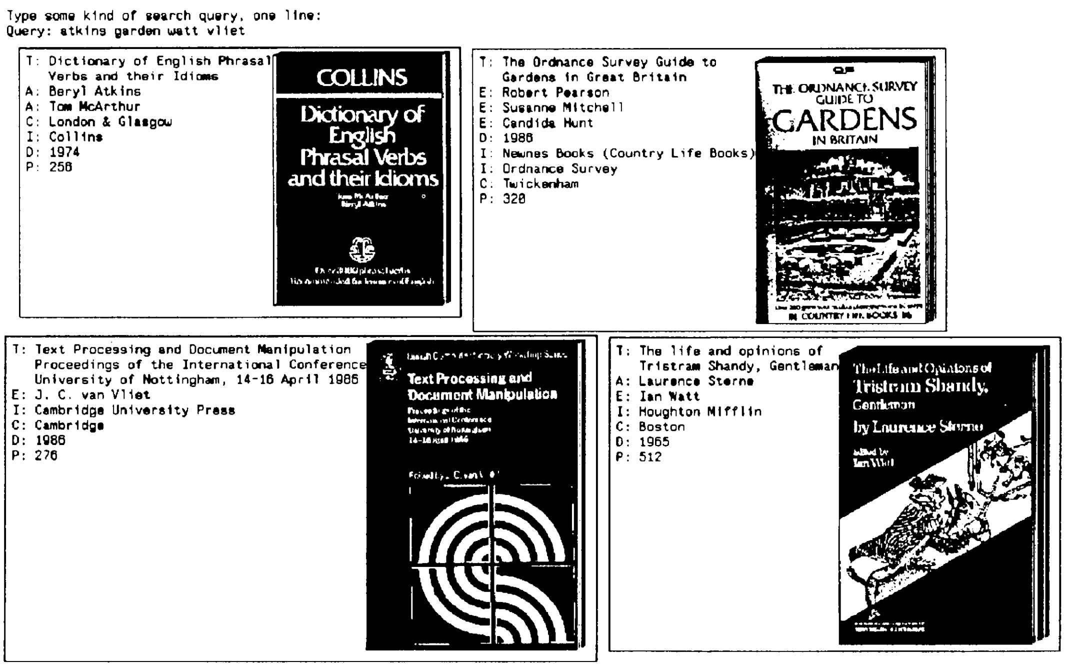


Figure 2.

On balance the comment that the dust jacket pictures use 75% of the screen space and 95% of the disk space to display no extra information over the book citation seems valid. A sample full screen with graphics is shown in Figure 3,

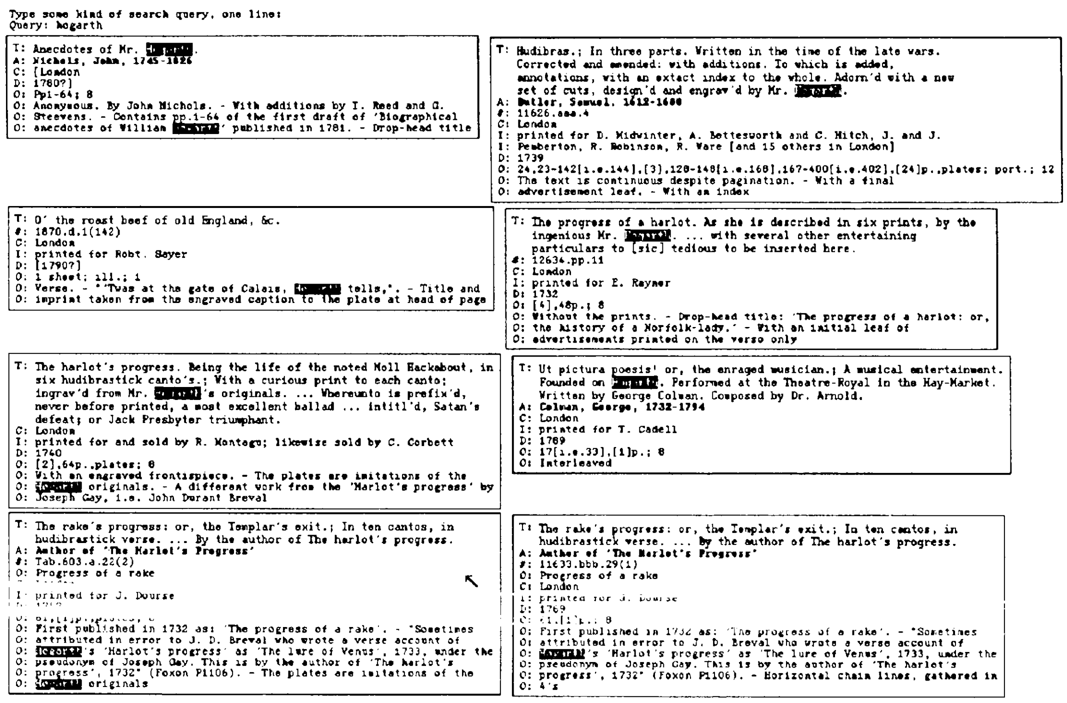


```
Type some kind of search query, one line:
Query: atkins garden watt vliet

T: Dictionary of English Phrasal
   Verbs and their Idioms
A: Beryl Atkins
A: Tom McArthur
C: London & Glasgow
I: Collins
D: 1974
P: 256

T: The Ordnance Survey Guide to
   Gardens in Great Britain
E: Robert Pearson
E: Susanne Mitchell
E: Candida Hunt
D: 1986
I: Newnes Books (Country Life Books)
I: Ordnance Survey
C: Twickenham
P: 320

T: Text Processing and Document Manipulation
   Proceedings of the International Conference
   University of Nottingham, 14-16 April 1986
E: J. C. van Vliet
I: Cambridge University Press
C: Cambridge
D: 1986
P: 276

T: The life and opinions of
   Tristram Shandy, Gentleman
A: Laurence Sterne
E: Ian Watt
I: Houghton Mifflin
C: Boston
D: 1965
P: 512
```

COLLINS
Dictionary of English Phrasal Verbs and their Idioms
THE ORDNANCE SURVEY GUIDE TO GARDENS IN BRITAIN
Text Processing and Document Manipulation
The Life and Opinions of Tristram Shandy, Gentleman
by Laurence Sterne

Figure 3.

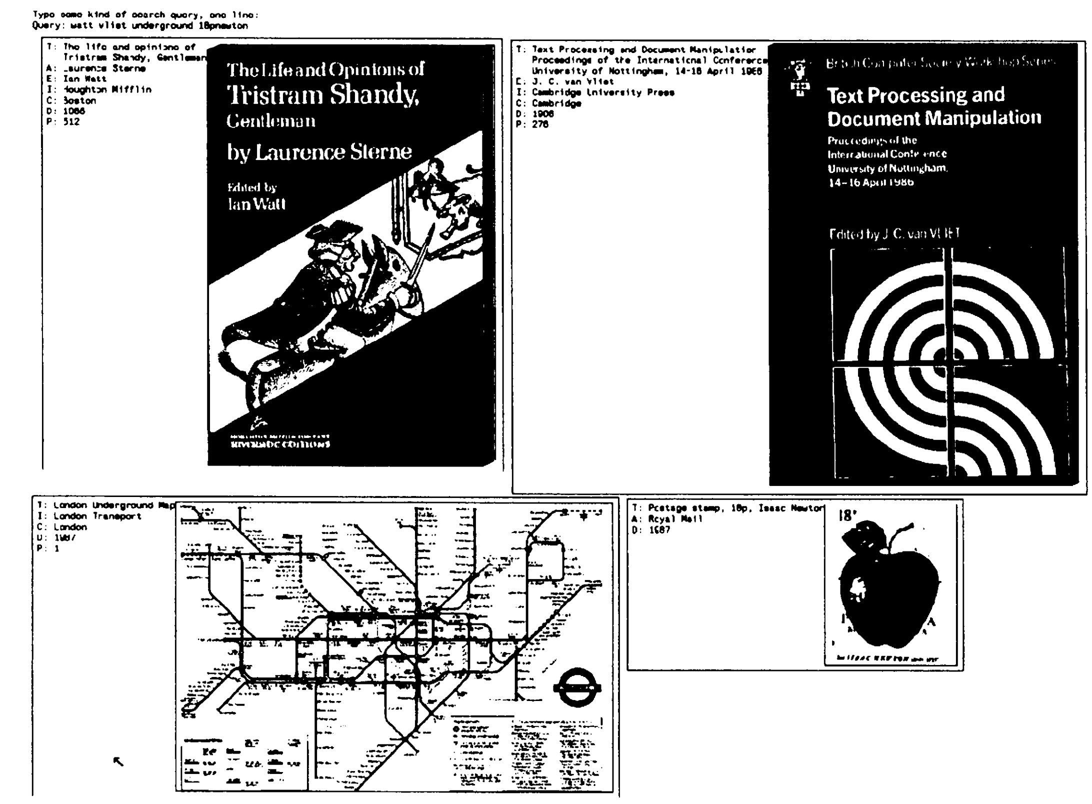
and a sample screen with a more normal display (in this case items relating to Hogarth) is shown in Figure 4.



```
Type some kind of search query, one line:
Query: hogarth

T: Anecdotes of Mr. Hogarth.
A: Nichols, John, 1745-1826
C: [London
D: 1780?]
O: Ppi-64; 8
O: Anonymeous. By John Michols. - With additions by I. Reed and G.
O: Steevens. - Contains pp.1-64 of the first draft of 'Biographical
O: anecdotes of William Hogarth' published in 1781. - Drop-head title

T: Hudibras.; In three parts. Written in the time of the late wars.
   Corrected and amended: with additions. To which is added,
   annotations, with an extact index to the whole. Adorn'd with a new
   set of cuts, design'd and engrav'd by Mr. Hogarth.
A: Butler, Samuel, 1612-1680
#: 11626.aaa.4
C: London
I: printed for D. Midwinter, A. Bettesworth and C. Hitch, J. and J.
I: Pemberton, R. Robinson, R. Ware [and 15 others in London]
D: 1739
O: 24,23-142[i.e.144],[3],128-148[i.e.168],167-400[i.e.402],[24]p.,plates; port.; 12
O: The text is continuous despite pagination. - With a final
O: advertisement leaf. - With an index

T: O' the roast beef of old England, &c.
#: 1870.d.1(142)
C: London
I: printed for Robt. Sayer
D: [1790?]
O: 1 sheet; ill.; 1
O: Verse. - "'Twas at the gate of Calais, Hogarth tells,". - Title and
O: imprint taken from the engraved caption to the plate at head of page

T: The progress of a harlot. As she is described in six prints, by the
   ingenious Mr. Hogarth. ... with several other entertaining
   particulars to [sic] tedious to be inserted here.
#: 12634.pp.11
C: London
I: printed for E. Rayner
D: 1732
O: [4],48p.; 8
O: Without the prints. - Drop-head title: 'The progress of a harlot: or,
O: the history of a Norfolk-lady.' - With an initial leaf of
O: advertisements printed on the verso only

T: The harlot's progress. Being the life of the noted Moll Hackabout, in
   six hudibrastick canto's.; With a curious print to each canto;
   ingrav'd from Mr. Hogarth's originals. ... Whereunto is prefix'd,
   never before printed, a most excellent ballad ... intitl'd, Satan's
   defeat; or Jack Presbyter triumphant.
C: London
I: printed for and sold by R. Montagu; likewise sold by C. Corbett
D: 1740
O: [2],64p.,plates; 8
O: With an engraved frontispiece. - The plates are imitations of the
O: Hogarth originals. - A different work from the 'Harlot's progress' by
O: Joseph Gay, i.e. John Durant Breval

T: Ut pictura poesis! or, the enraged musician.; A musical entertainment.
   Founded on Hogarth. Performed at the Theatre-Royal in the Hay-Market.
   Written by George Colman. Composed by Dr. Arnold.
A: Colman, George, 1732-1794
C: London
I: printed for T. Cadell
D: 1789
O: 17[i.e.33],[1]p.; 8
O: Interleaved

T: The rake's progress: or, the Templar's exit.; In ten cantos, in
   hudibrastick verse. ... By the author of The harlot's progress.
A: Author of 'The Harlot's Progress'
#: Tab.603.a.22(2)
O: Progress of a rake
C: [illegible]
I: printed for J. Dourse
D: [illegible]
O: [illegible]
O: First published in 1732 as: 'The progress of a rake'. - "Sometimes
O: attributed in error to J. D. Breval who wrote a verse account of
O: Hogarth's 'Harlot's progress' as 'The lure of Venus', 1733, under the
O: pseudonym of Joseph Gay. This is by the author of 'The harlot's
O: progress', 1732" (Foxon P1106). - The plates are imitations of the
O: Hogarth originals

T: The rake's progress: or, the Templar's exit.; In ten cantos, in
   hudibrastick verse. ... By the author of The harlot's progress.
A: Author of 'The Harlot's Progress'
#: 11633.bbb.29(1)
O: Progress of a rake
C: London
I: printed for J. Dourse
D: 1769
O: 41,[1]p.; 8
O: First published in 1732 as: 'The progress of a rake'. - "Sometimes
O: attributed in error to J. D. Breval who wrote a verse account of
O: Hogarth's 'Harlot's progress' as 'The lure of Venus', 1733, under the
O: pseudonym of Joseph Gay. This is by the author of 'The harlot's
O: progress', 1732" (Foxon P1106). - Horizontal chain lines, gathered in
O: 4's
```

Figure 4.

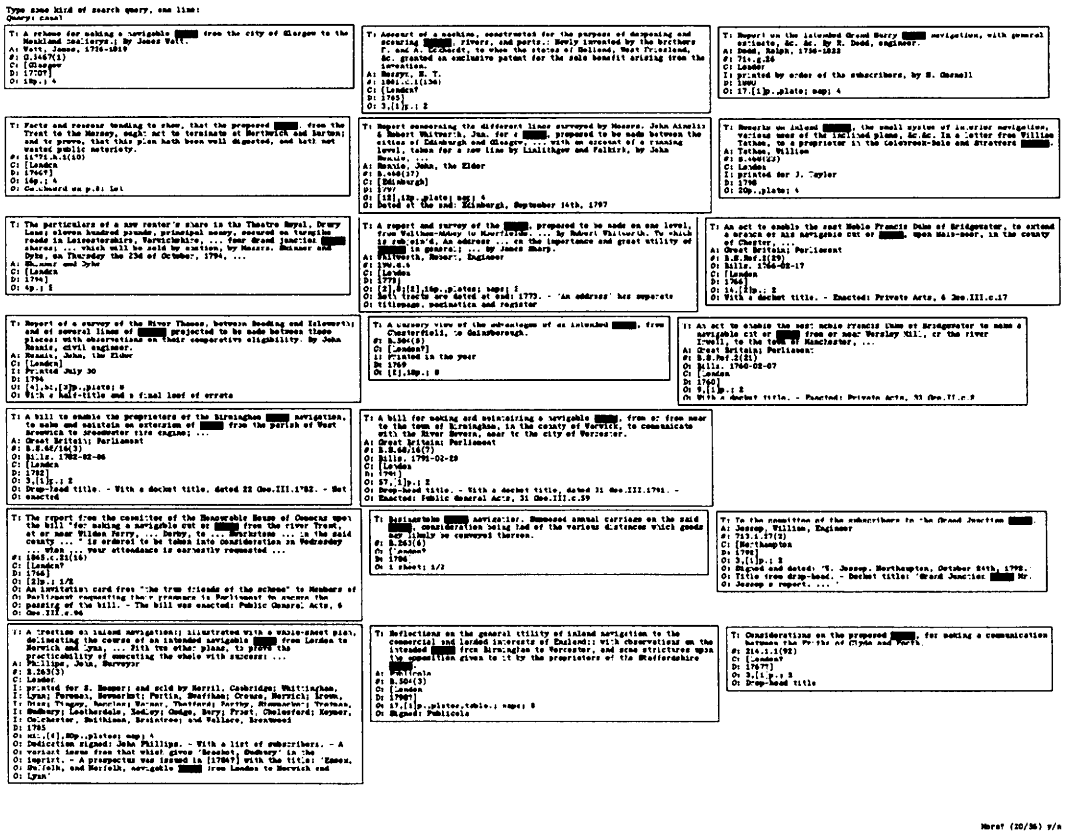
With a high resolution Sun screen even more detail can be produced: here are two more sample figures showing the results. Compare, for example, the covers of the books by Sterne and Van Vliet which are in both figures 3 and 5. Note that even with the 1500x1200 screen, there is still a noticeable lack of readability for fine print in the display. For example, in the London Underground map the detail to read the labels is simply not there in a 75 dpi digitization of this well printed original; similarly the 18p postage stamp (originally in color) is not clear enough. Displaying only one dust jacket at a time solves that problem (by permitting use of a 150 dpi digitization), but makes scanning the catalog so slow that it is undesirable.



```
Typo somo kind of search query, one line:
Query: watt vliet underground 18pnewton

T: The life and opinions of
   Tristram Shandy, Gentleman
A: Laurence Sterne
E: Ian Watt
I: Houghton Mifflin
C: Boston
D: 1066
P: 512

T: Text Processing and Document Manipulation
   Proceedings of the International Conference
   University of Nottingham, 14-18 April 1966
E: J. C. van Vliet
I: Cambridge University Press
C: Cambridge
D: 1906
P: 276

T: London Underground Map
I: London Transport
C: London
D: 1987
P: 1

T: Postage stamp, 18p, Isaac Newton
A: Royal Mail
D: 1987
```

Figure 5.

And here is the amount of catalog that can be displayed using a high resolution screen and relatively small fonts. This is almost as much of an eye test on the screen as in this paper but note that in real life the British Library catalog does include this much information on each page; of course the pages are somewhat larger, and the formatting more elegant.



```
Type some kind of search query, one line:
Query: canal

T: A scheme for making a navigable canal from the city of Glasgow to the
   Monkland Coalierys.; By James Watt.
A: Watt, James, 1736-1819
#: G.3467(1)
C: [Glasgow
D: 1770?]
O: 18p.; 4

T: Account of a machine, constructed for the purpose of deepening and
   scouring canals, rivers, and ports.: Newly invented by the brothers
   F. and A. Eckhardt, to whom the states of Holland, West Friesland,
   &c. granted an exclusive patent for the sole benefit arising from the
   invention.
A: Hessyn, H. T.
#: 1061.c.1(136)
C: [London?
D: 1785]
O: 3,[1]p.; 2

T: Report on the intended Grand Surry canal navigation, with general
   estimate, &c. &c. By R. Dodd, engineer.
A: Dodd, Ralph, 1756-1822
#: 714.g.26
C: London
I: printed by order of the subscribers, by S. Gosnell
D: 1800
O: 17,[1]p.,plate; map; 4

T: Facts and reasons tending to shew, that the proposed canal, from the
   Trent to the Mersey, ought not to terminate at Northwick and Barton;
   and to prove, that this plan hath been well digested, and hath not
   wanted public notoriety.
#: 11771.h.1(10)
C: [London
D: 1766?]
O: 16p.; 4
O: Catchword on p.8: Let

T: Report concerning the different lines surveyed by Messrs. John Ainslie
   & Robert Whitworth, Jun. for a canal, proposed to be made between the
   cities of Edinburgh and Glasgow, ... with an account of a running
   level, taken for a new line by Linlithgow and Falkirk, by John
   Rennie, ...
A: Rennie, John, the Elder
#: B.460(17)
C: [Edinburgh]
D: 1797
O: [12],12p.,plate; map; 4
O: Dated at the end: Edinburgh, September 14th, 1797

T: Remarks on inland canals, the small system of interior navigation,
   various uses of the inclined plane, &c.&c. In a letter from William
   Tatham, to a proprietor in the Colebrook-Dale and Strafford canal.
A: Tatham, William
#: B.460(23)
C: London
I: printed for J. Taylor
D: 1798
O: 20p.,plate; 4

T: The particulars of a new renter's share in the Theatre Royal, Drury
   Lane; eleven hundred pounds, principal money, secured on turnpike
   roads in Leicestershire, Warwickshire, ... four Grand Junction canal
   shares; ... which will be sold by auction, by Messrs. Skinner and
   Dyke, on Thursday the 23d of October, 1794, ...
A: Skinner and Dyke
C: [London
D: 1794]
O: 4p.; 2

T: A report and survey of the canal, prepared to be made on one level,
   from Waltham-Abbey to Moorfields. ... By Robert Whitworth. To which
   is subjoin'd, An address ... on the importance and great utility of
   canals in general; ... by James Sharp.
A: Whitworth, Robert, Engineer
#: 190.d.6
C: [London
D: 1773]
O: [2],8,[2],16p.,plates; maps; 2
O: Both tracts are dated at end: 1773. - 'An address' has separate
O: titlepage, pagination and register

T: An act to enable the most Noble Francis Duke of Bridgewater, to extend
   a branch of his navigable cut or canal, upon Hale-moor, in the county
   of Chester, ...
A: Great Britain; Parliament
#: B.S.Ref.2(29)
O: Bills. 1766-02-17
C: [London
D: 1766]
O: 14,[2]p.; 2
O: With a docket title. - Enacted: Private Acts, 6 Geo.III.c.17

T: Report of a survey of the River Thames, between Reading and Isleworth;
   and of several lines of canal projected to be made between those
   places: with observations on their comparative eligibility. By John
   Rennie, civil engineer.
A: Rennie, John, the Elder
C: [London]
I: Printed July 30
D: 1794
O: [4],52,[3]p.,plate; 4
O: With a half-title and a final leaf of errata

T: A cursory view of the advantages of an intended canal, from
   Chesterfield, to Gainsborough.
#: B.504(8)
C: [London?]
I: Printed in the year
D: 1769
O: [2],18p.; 8

T: An act to enable the most noble Francis Duke of Bridgewater to make a
   navigable cut or canal from or near Worsley Mill, cr the river
   Irwell, to the town of Manchester, ...
A: Great Britain; Parliament
#: B.S.Ref.2(21)
O: Bills. 1760-02-07
C: [London
D: 1760]
O: 9,[1]p.; 2
O: With a docket title. - Enacted: Private Acts, 33 Geo.II.c.2

T: A bill to enable the proprietors of the Birmingham canal navigation,
   to make and maintain an extension of canal from the parish of West
   Bromwich to Broadwater fire engine; ...
A: Great Britain; Parliament
#: B.S.68/16(3)
O: Bills. 1782-02-06
C: [London
D: 1782]
O: 3,[1]p.; 2
O: Drop-head title. - With a docket title, dated 22 Geo.III.1782. - Not
O: enacted

T: A bill for making and maintaining a navigable canal, from or from near
   to the town of Birmingham, in the county of Warwick, to communicate
   with the River Severn, near to the city of Worcester.
A: Great Britain; Parliament
#: B.S.68/16(7)
O: Bills. 1791-02-28
C: [London
D: 1791]
O: 57,[1]p.; 2
O: Drop-head title. - With a docket title, dated 31 Geo.III.1791. -
O: Enacted: Public General Acts, 31 Geo.III.c.59

T: The report from the committee of the Honourable House of Commons upon
   the bill "for making a navigable cut or canal from the river Trent,
   at or near Wilden Ferry, ... Derby, to ... Burslem ... in the said
   county ... " is ordered to be taken into consideration on Wednesday
   ... when ... your attendance is earnestly requested ...
#: 1865.c.21(16)
C: [London?
D: 1766]
O: [2]p.; 1/2
O: An invitation card from "the true friends of the scheme" to Members of
O: Parliament requesting their presence in Parliament to secure the
O: passing of the bill. - The bill was enacted: Public General Acts, 6
O: Geo.III.c.96

T: Basingstoke canal navigation. Supposed annual carriage on the said
   canal, consideration being had of the various distances which goods
   may likely be conveyed thereon.
#: B.263(6)
C: [London?
D: 1796]
O: 1 sheet; 1/2

T: To the committee of the subscribers to the Grand Junction canal.
A: Jessop, William, Engineer
#: 713.1.27(2)
C: [Northampton
D: 1792]
O: 3,[1]p.; 2
O: Signed and dated: 'W. Jessop. Northampton, October 24th, 1792.'
O: Title from drop-head. - Docket title: 'Grand Junction canal Mr.
O: Jessop's report. ... '

T: A treatise on inland navigation;; illustrated with a whole-sheet plan,
   delineating the course of an intended navigable canal from London to
   Norwich and Lynn, ... With two other plans, to prove the
   practicability of executing the whole with success: ...
A: Phillips, John, Surveyor
#: B.263(3)
C: London
I: printed for S. Hooper; and sold by Merril, Cambridge; Whittingham,
I: Lynn; Peroman, Newmarket; Fortin, Swaffham; Crouse, Norwich; Brown,
I: Diss; Tingey, Beccles; Warner, Thetford; Forthy, Stowmarket; Trotman,
I: Sudbury; Loothardale, Hadley; Gedge, Bury; Frost, Chelmsford; Keymer,
I: Colchester, Smithison, Braintree; and Wallace, Brentwood
D: 1785
O: xii,[4],80p.,plates; map; 4
O: Dedication signed: John Phillips. - With a list of subscribers. - A
O: variant issue from that which gives 'Brooket, Sudbury' in the
O: imprint. - A prospectus was issued in [1784?] with the title: 'Essex,
O: Suffolk, and Norfolk, navigable canal from London to Norwich and
O: Lynn'

T: Reflections on the general utility of inland navigation to the
   commercial and landed interests of England;: with observations on the
   intended canal from Birmingham to Worcester, and some strictures upon
   the opposition given to it by the proprietors of the Staffordshire
   canal.
A: Publicola
#: B.504(3)
C: [London
D: 1790?]
O: 17,[1]p.,plates,table.; maps; 8
O: Signed: Publicola

T: Considerations on the proposed canal, for making a communication
   between the Friths of Clyde and Forth.
#: 214.1.1(92)
C: [London?
D: 1767?]
O: 3,[1]p.; 2
O: Drop-head title

                                                          More? (20/36) y/n
```

Figure 6.

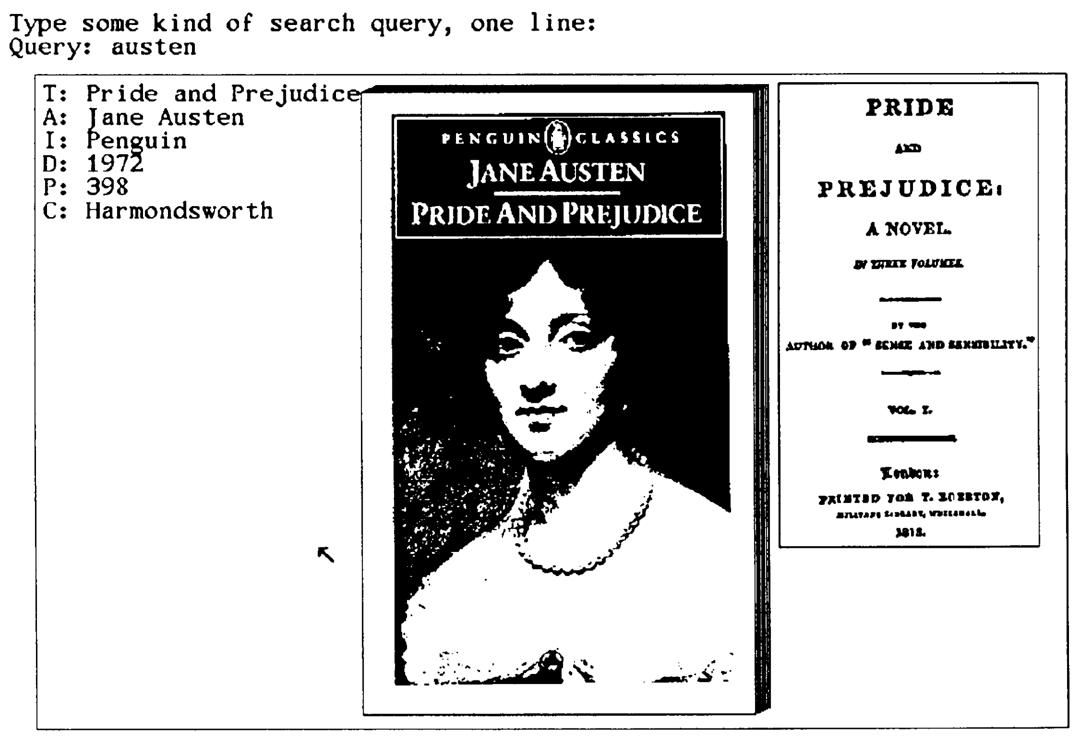
It is also possible, where the full text of the book is stored, to go directly to reading the text. Thus, for *Pride and Prejudice* the possible levels of output might be:

```
Pride and Prejudice / Jane Austen / 1972
```

```
T: Pride and Prejudice
A: Jane Austen
I: Penguin
D: 1972
P: 398
C: Harmondsworth
```

---

then the picture of the book (see Figure 7),



```
Type some kind of search query, one line:
Query: austen
T: Pride and Prejudice
A: Jane Austen
I: Penguin
D: 1972
P: 398
C: Harmondsworth
```

Figure 7.

and finally the text:

```
<T PRIDE AND PREJUDICE><V I><C I>
IT is a truth universally acknowledged, that a single man in
possession of a good fortune, must be in want of a wife.
However little known the feelings or views of such a man
may be on his first entering a neighbourhood, this truth is so
well fixed in the minds of the surrounding families, that he is
considered as the rightful property of some one or other of
their daughters.
"My dear Mr. Bennet,' said his lady to him one day, "have
you heard that Netherfield Park is let at last?'
...
```

Two works are available for demonstration in this form.

Search Vocabulary Improvements

Two specific features of the new software not commonly found in online catalogs seem to work well: term expansion and phrase presentation. Term expansion is aimed at the problems of low recall. Should the user type a word that is not found, the system looks in its dictionaries and tries to find overlapping words. For example, *cider* is defined as *fermented apple juice*. Looking for other words defined with *ferment*, *apple*, and *juice*, the program finds that *wine* is defined as *alcoholic drink made from the fermented juice of grapes* and *mead* is defined as *alcoholic drink made from fermented honey and water*. So when there are not enough items found using the word *cider* the program expands it to

```
Query: cider
   Expanding to:  apple ciderpress ferment juice sour tartar wine mead perry pulque
```

and then retrieves, for example:

```
T: A treatise on the culture of the apple & pear, and on the manufacture
   of cider & perry. By T. A. Knight ...
A: Knight, Thomas Andrew
#: 988.c.27
C: Ludlow
I: printed and sold by H. Procter; sold also by T. N. Longman, London;
I: the booksellers in Hereford, Leominster, Worcester &c. &c.
D: 1797
O: 162,XXIII,[3]p.; 8
O: With a final errata leaf

T: The true amazons:; or, the monarchy of bees. Being a new discovery and
   improvement of those wonderful creatures. ... Also how to make the
   English wine or mead, ... By Joseph Warder ...
A: Warder, Joseph
C: London
I: printed for John Pemberton; and William Taylor
D: 1722
O: 16,v-xiii,[3],120p.,plate; port.; 8
O: A reissue of the fourth edition, with a new titlepage followed by
O: 14pp. section 'To the booksellers'
```

The other problem, that of low precision, is dealt with by suggesting phrases. Suppose, for example, one types the word **heart** as the search term. This would retrieve 145 items. The program prints out a table of adjoining words and asks if you like any of them:

```
    Here are some words that abut the word you used
Can I interest you in  or in..
 loyal heart (2)        heart lover (2)
 broken heart (5)       heart ache (3)
 simple heart (2)       heart intend (3)
 true heart (4)         heart woman (2)
 light heart (2)        heart trouble (2)
 bleed heart (2)        heart written (5)
 sweet heart (3)
 own heart (3)
 hard heart (2)
    Mark (with right mouse button) or type the words you want added,
    then hit return
```

Suppose you mark *broken heart, hard heart, heart ache and heart trouble*. This retrieves 11 items. Some of them are what you would expect:

```
T: The broken hearted lover's garland; containing five new songs. I. The
   broken hearted lover. II. The young man's earnest request to fair
   Cloie. ... III. The batchelor's lamentation, ... IV. Nell's
   constancy. V. The seaman's answer. ...
#: 11621.b.60(5)
C: [Newcastle?
D: 1750?]
O: 8p.; 12
```

or

```
T: A cure for the heart-ache;; a comedy, in five acts, as performed at
   the Theatre-Royal, Covent-Garden. By Thomas Morton, ...
A: Morton, Thomas, 1764-1838
#: 643.f.7(6)
C: London
I: printed for T. N. Longman
D: 1797
O: 87,[1]p.; 8
```

but others are perhaps a bit surprising:

```
T: A rueful story or Britian in tears, being the conduct of Admiral B-g,
   in the late engagement off Mahone, with a French fleet the 20. of
   May. 1756
#: T.1070(1*)
C: London
I: printed by Boatswain Hawl-Up. a broken hearted sailor
D: [1756]
O: [1],6-15,[1]p.; 8
```

Implementation

The UNIX system was particularly useful for implementing this system for several reasons. The most important was the existence of many of the utilities needed. For example, some of the large data files (in particular the catalog itself and the dictionaries) need access by each word. So a database library is needed to record the locations in the file of each different word. For this Peter Weinberger's B-tree library[8] was suitable, since it handles very large files efficiently, is easy to use and robust, and provides fast retrieval. Other databases, however, did not require quite such flexibility in storage: e.g. the file of the number of occurrences of each word, used to decide on the weighting factor in expansion. So that file was stored using an extensible hashing package by Michael Hawley of MIT, which is more compact than a B-tree. Other experiments were made also: for example, if storage space had been even tighter, there was a system for using compressed bitvectors to provide slower retrieval with less storage overhead. In practice, however, the price of disk space is declining fast enough to make the B-tree system practical. What was particularly convenient about UNIX is that all three of these retrieval systems existed with sufficiently similar interfaces and sufficiently small implementations that all could be loaded and the code switched easily from one to another as space or time became critical quantities.

[8] P. J. Weinberger, *UNIX Programmer's Manual, Eighth Edition*, AT&T Bell Laboratories, Murray Hill, New Jersey, February 1985. See section cbt(3).

Similarly, it was necessary to have a window manager which could display graphics and multiple fonts on a large screen. Readability is much better on large screens, particularly with a library catalog where one wishes to display as many items as possible. Steven Uhler's MGR manager was very convenient for this. Again, it was not necessary to spend weeks studying the manuals for this system to just get simple pictures of dust jackets and multiple font displays working. And again, there were other window managers that could have been used had this not been available.

From the standpoint of eventual use by a final customer (not yet established, as this is a research project not yet adopted by anybody), it is also an advantage to have implemented this on UNIX because of the wide availability of UNIX systems on different size hardware. Some libraries prefer large mainframes for their OPACs (often they have them, or have access to them, already for other reasons); some prefer minicomputers; and some hope to implement their catalogs on micros. UNIX runs in all these environments and thus eliminates the need to argue about equipment scale.

Comments

There were other features tried with less success. These included relevance feedback (it will be necessary to stipulate whether the feedback is on all fields of the citation, or only on the title field), not suffixing the words, and attempting sense disambiguation. The greatest disappointment was that I expected to be able to use the BL pressmark field for searching; having identified one book in a given subject, it would seem reasonable to then browse the shelflist. But the pressmark system was changed over time, and has so many idiosyncrasies, and is so unfamiliar to the average user, that would be difficult to explain how to use these pressmarks as subject searches.

It is more important than I originally thought to consider in which field of the record the words appeared. There were a surprising number of strange retrievals caused by comments on binding, printers names, and the like. This is, I suspect, a property of the loquacity of eighteenth century title pages and would be less serious with modern records.

The lessons from this work are that (a) it isn't all that difficult to produce a demonstration of an online catalog; (b) it is useful to try term expansion; (c) it is useful to dynamically present possible phrases; and (d) dictionaries can help with the problems of vocabulary shifts over centuries. And, it is convenient to use a UNIX system for developing this sort of program. For possible next steps, it would be very valuable to have more information about individual books; it would be interesting to compare the effectiveness of term expansion with a thesaurus to the dictionary program used here; and I await the OED in machine-readable form to help with the changes in vocabulary over time and to give more information about word uses.

Acknowledgments: My thanks to the Oxford Text Archive, Collins Publishers, Oxford University Press, the British Library, the Eighteenth Century Short Title Catalog (ESTC), and the University of Waterloo for assistance and tapes of dictionaries and catalogs.

Software Tools for Music
- or -
Communications Standard Works!

David Keeffe

Siemens Ltd., Systems Development Group,
Woodley, Reading, UK
ukc!siesoft!dk

1. Introduction

Described here is the evolution of a small suite of programs for the composition and performance of music. They started life as a personal interest, inspired in part by Peter Langston's work [Langston 86]. As they developed, however, a use for the programs was seen as an unusual and illustrative aid for exhibiting computing equipment.

At the Systems Development Group, we are involved variously in developing systems which address the problems of UNIX and DOS communication, in developing software for a graphics workstation, and generally in improving the flexibility and usability of Siemens range of UNIX and DOS machines. Would it not then be a good idea if an ''earcatching'' package could be built which combined all this?

As such a system is to receive close scrutiny, the musical ideas must have a reasonable foundation: the computer should not be seen as simply a glorified tape machine.

The aim of this paper, then, is to present several facets of the music system: of course, the musical ideas are central, but there are also other lessons to be learnt. There are two central foundations of the design of the system: the first is the Musical Instrument Digital Interface, or MIDI: there will not be much said about it, as there isn't much to say — the standard itself is only about one-third the length of this paper! What should become clear is not only the way MIDI allows the system to function but also how such a useful standard can make writing other music programs so much easier and more effective. The second foundation is the legacy of the UNIX operating system: it is that legacy which makes the whole thing fit together.

Why music? The composition of music is generally thought of as one of the most abstract of human activities. While not ever hoping to replace the human musician, the computer can be used to experiment with music, as well as providing a base for a diverting exercise in analysis and programming. Also, computer music is strangely attractive, like high-quality intelligent speech systems.

For many people it is summed up by Dr. Johnson's comment [Johnson]: ''...[it] is like a dog walking on his hinder legs. It is not done well; but one is surprised that it is done at all.''

2. Music Tools for UNIX

We will now focus on the musical problems; the consequent generalities about programming and communication are best discussed once the specific tasks have been presented. That is, where now the *music* side is the point of view, later the UNIX and communications side will provide a standpoint (''UNIX Tools for Music'').

2.1. A Little History

There have been many experiments and products over the years which combine computers and music. The computer has been used as a sound generation system (eg MUSIC V, Music 360) and as a compositional tool. Admittedly the widest audience has been for computers as performers, but nevertheless some modern computer-composed music has had general acceptance. In particular Iannis Xenakis has composed several works with computer-generated sections.

The main problem of old was either the great cost of hardware, or, as costs reduced, the lack of any commonly accepted means of defining and producing something that could be played or heard. If the computer was a composition tool, then suitable output had to be produced for human players or for some other program to use. On the other hand, performance programs (like MUSIC V) had their own language, and no ready means of accepting direct input, say, from a piano keyboard. In the case of microcomputers direct calculated digital sound was impractical and special hardware had to be used: usually the limitation here was that the number and quality of the sounds producible was severely limited.

Alongside the computing community, the musical world was getting increasingly frustrated by the growing plethora of different electronic instruments and the corresponding difficulties in using them together. Even five years ago, it was not unusual for audiences of some rock bands to see two or three stacked banks of different keyboard instruments and a hyperactive player leaping between them. Clearly something had to be done.

2.2. MIDI

Some musical instrument manufacturers (mostly Japanese) had invented means of interconnecting their own instruments, thereby easing some burdens on the musician. It seemed that the "video standard" problem was about to repeat itself, but in 1985 [MIDI 1985], several US and Japanese manufacturers buried their differences and agreed on a communications standard for music. What was more, the standard had to define a system that could be used by self-confessed mechanical idiots: only some musicians take pride in their electronic prowess — most baulk at having to insert more than a handful of plugs!

The Musical Instrument Digital Interface (or MIDI) standard defines everything from the electrical characteristics, through the wiring of plugs to the data formats to be used. There are even basic circuits supplied in the definition! The standard itself runs to about 6 pages. The first release of MIDI (1.0) has been proved eminently practical and successful[1]. What other inter-manufacturer standard consistently allows one to buy any two pieces of equipment, plug them together using a lead bought from the corner shop, and immediately to start using them?

It was the existence of MIDI which encouraged the present work — indeed without it, it is safe to say nothing would have been done! MIDI not only provided a means of controlling a high-quality synthesiser, but also gave a solid basis for the internal storage and data structures used by the composition programs. Since a finite set of controllable parameters is defined by MIDI, then aspects of composition and performance outside these bounds need not be addressed: a large chunk of design and planning was rendered unnecessary.

2.3. Music Hardware

What machinery is used for this work? There is an absolute minimum set and there is the set actually used by the demonstration package.

In both cases the total hardware involved is relatively simple: the demonstration set is listed below. Only the OP-4001 and the TX81Z are directly concerned with music with a total cost of about 600 pounds. Cheaper synthesisers are available.

1. Siemens X20 graphics workstation (UNIX OS)
2. Siemens PC-D2 (PC AT clone)
3. Voyetra Technologies OP-4001 midi card for IBM PC
4. Technics SX-PX7 Digital Piano
5. Roland MT-32 Synthesiser Module
6. Yamaha TX81Z Synthesiser Module

Of the computing equipment, perhaps the X20 is the least familiar: it is an NSC32016-based UNIX workstation, running a multiple-universe version of UNIX. It supports NFS.

The Voyetra OP-4001 is an intelligent MIDI interface, taking responsibility for filtering and timing messages onto the MIDI bus.

The synthesiser modules both have the advantage that they can act as if they were in effect several synthesisers: each is capable of playing different sounds simultaneously. In addition the MT-32 has a drum

1 Of course, like any new system, users rapidly try to drive it to its limits and then want the next version when they find them. Cries for MIDI 2.0 have been heard, but little seems to have happened! New functionality has been added recently (eg MIDI Time Code), but never at the expense of complete compatibility with existing equipment.

machine and reverberation unit.

The minimum set on which the original work was done consists of items 2, 3 and 6, with the PC-D2 replaced by an IBM PC Portable. However, all the programs (except one) were written to run on UNIX , but with DOS in mind. With the exception of the (crazy) need to distinguish 'text' from 'binary' on DOS they were completely source compatible with the X/Open standard. The one exception was the program to drive the OP-4001, which was and still is DOS-resident. There was no budget to develop an interface for UNIX.

The demonstration system mentioned above also incorporates an Ethernet. Music is composed on the X20, and then shipped using FTP or NFS to the PC which acts as a clever peripheral and plays music on the specified synthesisers. The X20 can be used to display conventional musical notation. All this will be expanded on later.

2.4. Music Software

The basic programming principle of the music software was 'keep it simple'. Because at times work had to be done on a PC, programs needed to be kept as small and conservative in programming style as possible.

The answer was to adopt the time-honoured UNIX principle of 'do just one thing and do it well'.

But in the case of music, what are those 'one things'? Isn't music an organic whole? Probably not: at least five different components can be identified:

1. melody — the sequential organisation of pitches
2. harmony — the simultaneous organisation of pitches
3. rhythm — the organisation of events in time
4. texture — the way simultaneous sequences of events interact other than harmonically
5. form — the large-scale sequential organisation of music.

Form could be seen as a type of rhythm: the main distinction is that 'rhythm' *per se* is heard as a single entity with little conscious use of memory, while 'form' requires the recognition of longer-term movement and repetition.

The intention of this project was to develop programs that gave the computer some control of these components. However, it was not clear at first which areas should merit closest attention. More questions needed asking and answering.

2.4.1. Musical Structure

First of all, it had to be decided what sort of music was going to be produced. If complex *avant-garde* music was the aim, then the scope for design was broadest, as it is really only a minority of listeners who can distinguish good from bad. However, it was decided to try to produce moderately pleasant and reasonably varied music suitable for background listening. To this end a guiding musical principle was proposed: "Keep at least one component (melody, harmony, rhythm, texture, form) of your music readily acceptable by most listeners." Before starting to develop programs to generate these musical components, there were these questions to be answered:

- why can computer music sound so boring?
- what is necessary to catch a listener's attention?
- how much should be left to the computer and how much predetermined?

The last question is easiest to start to answer: there seems little to discover or develop if the computer just plays back some pre-recorded music. The degree of control must lead to a feeling on the part of the listener that each composed piece is different in some way from the others, but still retaining some coherence and interest. What that degree should be can only be quantified by trying to answer the first two questions.

In a sense these questions are the same: by answering one, an answer is suggested for the other. Let us ask just one question, then, and suggest some answers.

What makes a piece of music start to be interesting? There are many things to consider, including the taste of the listener, but in general some rather coarse conclusions can be drawn.

- Completely unpredictable music is as unsatisfactory as completely predictable music.

- Keeping either or both of melody and rhythm completely predictable is unsatisfactory.
- Melody with unpredictable harmonic structure is unsatisfactory.
- Completely predictable harmony, form and texture can be satisfactory — witness the 12-bar blues, the classical passacaglia, the standard Verse-Chorus form of many folk songs, and the texture of the heavy metal band. Predictable harmony is only satisfactory if it has over about seven bars in the sequence.
- Unless great compositional care is taken, single-timbre music is less satisfactory than multiple-timbre music[2].

From this a basic recipe can be constructed:

1. Use static large-scale forms and textures. Since no piece is likely to exceed about two minutes in length, a standard 'A-B-A' structure will do.

   For texture, use a 4-player model (percussion, bass, accompaniment and solo) — this can be cut down if the musical style demands it: for example a pseudo-classical piece might not use the drummer!
2. Allow some variation in smaller-scale forms and textures: for example allow an introduction to an 'A' section in which the drummer starts, followed by say, the bass player.
3. Generate chord sequences randomly but according to constraining rules of progression. Some transitions could otherwise sound very strange! Realisation of the harmony as sound should not hide the progression: that is the basic notes of the chord must predominate.
4. Constrain rhythm to use an easily recognised metre and subdivision. In fact, all the music so far composed uses 4/4 time with no triplets!
5. Allow non-deterministic melody lines, but make it fit the current harmony and rhythm.

2.4.2. Music Tools

The current toolset presently consists of the following programs. Note that all except those for actual performance can run on any system with a suitable command interface and C compiler. Performance tools depend on the hardware available, but are probably partly portable across DOS-type microcomputers. A UNIX-based 'play' program would require a special device to cope with the timing constraints.

Performance

play	send MIDI file to MIDI controller

Composition

ddm	stochastic melody and rhythm generator
cgen	transition-matrix chord generator
flow	'choral' chord realiser
strum	'guitar' chord realiser
cread	chord symbol reader

Structuring

mrepeat	repeat bits of MIDI data
fixmidi	general MIDI transformation
mjoin	merge several MIDI tracks

Song Assembly

trkmerge	construct multitrack song file
mksong	make complete song from description

2 A timbre is a sound colour: for instance a guitar and a trumpet have different sounds even if they play the same music.

Debugging and Display

dumpmidi	display MIDI file as text
scribe	display MIDI file as musical score
chshow	display chord file as text

How did *this* set in its various categories come into being?

The first program was a driver for the OP-4001. This in turn demanded the design of a suitable file format to represent the music information: the note and dynamic information is MIDI-timing information is based on bars, beats and 'ticks' and interleaves with the MIDI messages. The playing program strips out the timing before sending the messages out on the MIDI bus.

The first composition program was an implementation of Peter Langston's 'Stochastic Binary Subdivision' (ddm) method of producing rhythms[1986]. This was used to fine-tune the MIDI file format as well as a first step in computer composition. Next came its extension to generate melodic lines also as defined by Langston. In both cases the music produced was insufficiently predictable to hold any lasting attention — only item 4 and part of item 5 of the recipe are being used[3].

It was time to impose some further structure: item 3 of the recipe demands some harmonic movement based on rules. Two programs were produced: one, 'cgen', uses a first-order transition matrix to generate harmonic progressions; the other, 'cread', simply translates chord symbols into a MIDI-based chord file. The matrix for **cgen** is a text file, and uses chord symbols and percentage probabilities. The user input to **cgen** defines not only the starting chord, the ending chord and the length of the sequence, but also intermediate 'milestones'. This allows users to have a little more control over context. Consideration has been given to (but as yet no use made of) higher-order transitions, but using contexts other than the preceding harmony, such as position in time or phrase structure.

Two new realisation tools came out of **cgen.** One ('flow') plays the chords as if sung or played on an organ, while the other ('strum') models a guitar and 'strums' the chords. To complete item 5 of the recipe, the **ddm** program was modified to understand the chords from **cgen** or **cread** and generate melodies to fit. This was done by restructuring the scale as defined in a **ddm** description file, to ensuring that the notes generated fitted the harmonic context. A further discovery was that if a **ddm** description contained little or no scope for relative melodic movement, the result was an animated but otherwise accurate realisation of the harmony itself. Slightly different contents in the **ddm** file are used to produce remarkably effective bass guitar lines.

Combination of harmonic realisation (of whatever form) and a **ddm** style melody increases musical interest dramatically. Three-part segments are even more appealing.

By now item 3 in the recipe has been achieved, and item 5 fully satisfied. The problem remaining is that the musical segments just start and stop without any further structure: once two or three have been heard, their appeal quickly diminishes. What is needed is form. Small-scale form can be achieved in various ways and at various levels: first to be addressed was a means by which complete multi-part segments have beginnings and ends. By modifying **flow, strum,** and **ddm** to accept the definition of where in time and in the harmony they were to start staggered entries and exits could be achieved. This allows texture to define an aspect of form. This goes some way to realising item 2 of the recipe.

In order to improve smaller-scale form the capability for structural constraint was introduced into **ddm.** Instead of inventing a fresh pattern for each bar, **ddm** was given a mechanism to repeat patterns and also to have empty bars. Because of the scale transformation algorithm, repeated bars have the same rhythm but do not necessarily repeat the exact melodic shape: this provides an interesting musical unpredictability.

The internal interest of musical segments has increased considerably, by restricting the variation and introducing longer rests. The problem remains that the bar/rest structure is fixed: a future project is to make the structure rule-derived.

So far only the generation of single segments of music has been described: better pieces will result if item 1 of the recipe is realised. As suggested, a basic 'A-B-A' structure has been used as the skeleton for larger-scale form. The second 'A' section will be an exact repeat of the first: subtle variations have been avoided as too difficult. The A-B-A structure may be refined in the medium-scale by introducing so-called

3 Some time later, a further facility was given to **ddm** to allow some form of syncopation to occur. The description file was extended to allow probabilities of omission of the first of a pair of beats, and between what subdivisions such omissions were to be allowed.

'bridges': thus a coarse A-B-A might become on closer inspection I-A-T-B-A-E where 'I' denotes 'introduction', 'T' denotes 'transition', and 'E' denotes 'ending'. Each segment can be generated in the same way as the main A and B: what is missing is a means of specifying how the segments are to be joined in time. A possible tool for this purpose would be a MIDI 'cat' program: however, because a MIDI file contains timing information, the **cat** operation would have to modify such values in the data stream before writing it out. Also, time-driven concatenation needs to know how to concatenate non-contiguous segments: section A might only have two bars of actual music in it, but actually need to last twenty: a section B must know where in time it is to be placed.

If we also consider form in a greater scale than that of the **ddm** 'form' specification but smaller than the 'A-B-A' under discussion we can see that each A or B might itself have an 'A-B-A' (or whatever) form. What is needed here is a) a tool which can repeat parts of a musical segment at some point in time, and b) a tool which can relocate the time location of a whole musical segment. The first tool, called 'mrepeat', allows the user to specify that a portion of the input file is to be replicated a number of times at a specified target point. Using this we can make a single 'A' become an 'A-?-A', or even change the shape of A itself. We might be able to make some subtle variation between the first and second instances of A. The second tool 'fixmidi' is a general purpose transformation tool which can move the time, pitch and timbre base of a single MIDI file. There is obviously some overlap between the function of **mrepeat** and **fixmidi** in the realm of time adjustment when the whole MIDI file is being manipulated: only **mrepeat** can move sections of a MIDI file. **Fixmidi** started life as a simple time-shifter, but was extended to allow fixed-interval pitch shifting and the insertion of MIDI program change messages. Since the basic action is always of copying a stream in, modifying it in some way and writing it out, it was thought more efficient to breach the 'single-function' guideline. Indeed **fixmidi** might be thought of as a precursor of a MIDI 'tr' or 'sed' program. Other transformations might be to modify the pitch of selected note values, to transpose in a given key, or to embed or filter MIDI controller messages.

We now have realised all of the items on the recipe except part of item 1. Although larger-scale forms can be imposed on single MIDI lines (or 'tracks') some means is needed to combine them; there are two possibilities: one is to merge two or more MIDI tracks into a new single-track file; the other is to collect a number of track files into one new multi-track file. Each has its uses, and in the end has the same effect on the set of MIDI instruments. The MIDI merging tool is called 'mjoin' and is like **cat** but ensures that all events are properly ordered in time, and significantly at the same time as they were in the unmerged files. The multi-tracking tool (called 'trkmerge') simply collects each specified MIDI file and stores then sequentially, with the addition of some header information. The difference is that a multitrack (or 'song') file allows track-by-track specification of MIDI information, such as basic channel. If a new routing is desired, then a simple modification of the headers can be undertaken.

What is left? We can construct musical segments with sufficient structure, and with sufficient interest to bear more than one listening — indeed they have a rather insistent memorability even if the musical content is not very inspired! The only difficulty is that much of the input is supplied by the user. The chord progression milestones have to be supplied and the matrix written, the **ddm** files have to be written and selected and the form has to be defined and constructed. In many respects this is a good state of affairs, as it ensures that the user exercises some musical influence. However, it is possible to leave more to the computer by having tools which generate the user input to the original set. Only one has been written at the time of this paper: it is a Bourne shell script which takes a user-written description file and produces a song file from it. No modification of the composition tools was necessary.

2.4.3. Summary and Future Work

From the work detailed above, it has become clear that more musical effect can be achieved by suitable structuring than by complex melody or rhythm generation. In fact, all melody and rhythm has been generated by a program based on Langston's 'Stochastic Binary Subdivision' principle, extended by chord following, constraints on structure, and syncopation. Although other mechanisms have been considered and experimented with, the most success has come from the tools for musical structuring.

What is to come? The 'scribe' program needs more work to be more than a graphics demonstration. Different style melodies and rhythms might be interesting: a fruitful path would be to separate the rhythmic structure from the pitch structure and then to 'fill in' the missing details using a separate tool. Indeed the **flow** program already has a handle for such information. There are also aspects and effects of **ddm** and **cgen** descriptions yet to be discovered.

3. UNIX Tools for Music

Having presented the rationale and development of a suite of programs for musical composition, it is reasonable to look at things from the other side: what is it that makes such a tool-based approach work, and what can be learnt?

First, let us consider what of the UNIX system has been used that makes the whole thing work.

3.1. The UNIX Legacy

As was mentioned before, all but two (at worst two) of the music programs described could run on . UX , DOS, VMS or indeed any operating system with a C compiler ''worth its salt''. The only programs which have any real dependency are **play,** which uses special hardware, **scribe** which depends on suitable graphical interface, and **mksong** which uses all sorts of features of the Bourne shell. However **mksong** should be capable of running on DOS or VMS if one of the implementations of the Bourne shell were used. What is clear is that this suite of programs does not depend on the UNIX operating system itself, but on the influence it has had on the computing world. Very few C compilers now offer a library which does not emulate the UNIX system. Most even allow UNIX -style command line interpretation and I/O redirection. Some even implement the semantics of pipes.

It is the author's contention that for most users, it is perceived behaviour which determines their model of the underlying system, rather than any intellectual understanding of its workings.[Keeffe 1984] It doesn't matter whether the programs here are running on DOS or UNIX : they were written to be portable across UNIX systems, and because of the UNIX legacy of a model of system behaviour, they work on other systems too. Again, the UNIX philosophy of small software tools forming an 'intersection' model of an operating system rather than than a 'union' model led to the almost inevitable design of a number of tools capable of working with each other in a number of ways [Atkins 87].

The original **cgen** program combined playable MIDI output with the calculation of chord layouts and the generation of suitable sequences. Very quickly an intersection was taken and **cgen** and **flow** were born. Because **cgen** produced a simple chord description, it could be used anywhere it was needed: similarly **flow** could accept chords from any file or program which produced the correct output. By using the **cread** program, even programs which produced text could be used.

Programs like **ddm** have a close interdependency between internally generated structures and external influences: it has been hard to separate out single functions. Still, by using **ddm** as a simple rhythm generator, other programs could be developed to add other components like pitch or harmony to the 'blank sheet' made available to them.

It is clear, then, that while little or none of the system described depends on the actual presence of a UNIX-based machine, without its existence the system would very likely have never been developed, or if it had been, it would have been in a far more restricted and less flexible form.

3.2. The Communications Legacy

3.2.1. MIDI as a Useful Standard

If the UNIX model determined the overall structure of the music system, it was a communications standard which determined much of how it worked. Because MIDI is a standard for use by completely non-technical people, it cannot have any part which even vaguely requires technical interpretation. How many times has a breakout box been needed to connect a simple VDU to a simple computer? MIDI does not need or even have breakout boxes.

Besides the ''plug and play'' benefits of MIDI, its strict range of values for a well-defined set of performance options saves the programmer the need to define such limits, and frees them for other work. The MIDI values are used from start to finish of the whole process, giving a constructional appeal to the music system due to the lack of any requirement to filter or transform the musical information. Yes, transformations are available, but they are information changers, not information preservers.

For example, chord descriptors use MIDI note numbers from the bottom octave (0-11) with octave offsets (multiples of 12) used to distinguish types of chord. This means that the realisation of chords simply uses the type information to choose a set of intervals, which are used to determine the actual notes of the chord.

3.2.2. Communications Problems

If MIDI provides an ideal base for the musical aspects of the system, what of non-musical communication? Will it prove as straightforward as connecting the synthesiser? Consider the demonstration package which uses the music system. Its aim is to present the following aspects of Siemens equipment:

1. the graphics capability of the X20 workstation
2. the flexibility of UNIX
3. communications between UNIX and DOS

A constraint on the demonstration is that it be operable by visitors to the stand at the show. To this end, it was decided that the DOS machine would operate automatically when a music demonstration was running.

The graphics workstation is eminently illustrated by converting MIDI information to musical notation using the 'scribe' program mentioned above. To many visitors, it is the actual presence of *music* which displays the flexibility of UNIX: they want applications flexibility, and are not interested in the underlying structure which makes combining of programs easy and successful. The ''hook'' is that they can actually hear what is produced. The problem to be solved was how the music files are made available to the PC for performance.

Several choices were available: if an Ethernet were available (which it was), then the DOS machine could run a control script which waited for FTP messages from UNIX , before running the control file so transferred — this depended on the provision of an FTP daemon and therefore on a particular supplier of Ethernet equipment; PC-NFS could be used if it was available, but would require some means of informing the DOS machine of the presence of a valid music file, and depended on a different supplier of Ethernet equipment. Without an Ethernet, a serial interface using a suitable communications package could be used: something like 'kermit' would do, albeit at a cost of speed, and at the cost of ensuring the cables actually worked properly. In the end, because suitable Ethernet facilities were available, both the FTP route and the NFS route were prepared: at the time of writing this paper one had not been selected.

The moral of this part of the tale is that if a standard like MIDI had been available, then the PC could have been connected to the UNIX machine with little effort: we are all told that such standards are coming 'real soon now' but when? and will they work so well?

4. Conclusions

It has been shown that a simple toolset addressing the problems of musical structuring can produce short ''songs'' of quite reasonable interest. By following a simple recipe attention has been given to the most important areas. It has also been shown that it is the UNIX legacy of programming and programming style rather than the presence of any particular UNIX system which has made the development of such a toolset relatively rapid and successful.

The existence of a single working music communications standard has made the whole thing take off: it has been contrasted to the continuing difficulties of connecting together two well-understood pieces of hardware.

5. References

[Langston 1986]
Peter Langston, ''(201) 644-2332 • Eedie and Eddie on the Wire, an Experiment in Music Generation'', EUUG Conference, Manchester 1986

[Johnson]
Life, Boswell

[Keeffe 1984]
''The Syntax of Operating System Command Languages'', DPhil Thesis, University of York, 1984

[MIDI 1985]
''MIDI 1.0 Detailed Specification'', International MIDI Association, US, 1985

''Atkins 1987''
Martin Atkins, Private Communication. (January 1987)

Adventures in UNIX Arithmetic

Robert Morris

National Computer Security Center
Fort George Meade
Maryland 20755
USA
RMorris@dockmaster.arpa

My first contact with the details of numerical software was in 1960 when a colleague came to me with a sad story asking for my help. It seemed that he had been doing some Fourier analysis and getting some unexpectedly noisy results. He tried some experiments to try to discover the cause of the problem and finally did what amounted to a Fourier transform of the library sine routine. He encountered a forest of high-frequency noise and came to me for help. Since, he said, the sine function does not have any high-frequency components, there must be something wrong either with his programs or with his mathematical understanding.

I took a look at the source code for the library sine routine, coded in 7090 machine language, and discovered that it used two separate approximations, one for angles less than 30 degrees, and another for larger angles. The discontinuity at 30 degrees was substantial and completely explained my colleagues anomalous results.

I have essentially never recovered from this friendly visit. I was surprised that such sloppy arithmetic had been tolerated so widely and for so long. I took a careful look, just for curiosity, at the other widely used mathematical library functions, and discovered that the problem with the sine routine was just the tip of the iceberg. For example, the routines for the Bessel functions returned the wrong sign for negative arguments and in some ranges, the values returned by the error function had no correct bits whatever.

Viewing this as a potentially very interesting problem, I recoded the entire math library for the 7090 to considerably higher standards of accuracy, a task that required some months, and then turned my attention to other matters, thinking that this problem was well under control.

It was, for the time, but over the years, as one computer environment was replaced by another, the same problem recurred. As these machines were supplied with essentially all system software supplied by the manufacturer, I did not have enough of the system under my control to do the job cleanly and correctly. For example, no part of the math library could in general be written in a high-order language, because the input conversion routines used by the compilers were themselves not accurate enough. The alternative was to write assembly-language routines and embed the necessary constants in machine floating-point representation. This approach would provide the required accuracy but it ensured that there would be absolute no possibility of porting all of this work to the next computer that came along.

With the development of the UNIX system, I had, for the first time, sufficient control over all of the arithmetic software to be able to do a decent job of providing accurate arithmetic and mathematical functions. I can take some of the credit for the mathematical software on UNIX, and, in some measure, my attempt to provide an accurate base of good arithmetic and accurate library functions was successful. At the same time I can take most of the blame for the ways in which UNIX software helps to provide inaccurate answers and other mathematical surprises.

One of my goals was to take as much of the magic as possible out of the construction of math library functions. Part of this was to find ways of coding the functions in a high-order language rather than in assembly language; another was to find a generic pattern for such programs which would make the programs as similar as possible to each other. The generic pattern has never emerged and there seem to be a large number of different patterns, almost equal in number to the number of basic routines. The sine and cosine routines, for example cannot be made to look anything like the logarithm and exponential routines. The square root function can easily be implemented with extreme accuracy and runs like the wind. The error function, on the other hand, is almost impossible to implement to any standard of accuracy whatever.

The other part of the attempt to take the magic out of math functions, that of coding them in a high-order language was considerably more successful. That depended on finding a small number of auxiliary functions to handle the few places where direct access to the machine floating-point registers was absolutely necessary. I finally settled on three auxiliary functions, called MODF, FREXP, and LDEXP. The first of these, MODF, was needed in some cases where there was a way to do the job in a high-order language, but there was no way to do it accurately enough. The latter two, FREXP, and LDEXP, were need to manipulate the exponent and fraction of the floating-point word separately, as required in the logarithm and exponential function. This choice has turned out to be a successful one, and is much more obviously correct in retrospect than it was at the time the decision had to be made.

It has also turned out to be possible to port the math library from machine to machine with considerable ease with little or no program modification. There are cases where the only changes that were need were to rewrite the three auxiliary functions to match a new instruction set or a different floating-point format.

One of the components that I needed in order to write accurate math library functions was some means of computing the necessary constants and coefficients to considerably greater accuracy than could fit in the registers of the target machine. It turns out to be roughly the case that to generate coefficients for an optimal polynomial or rational approximation to a function, you need to do the computation to about twice the number of digits as those available on the target machine.

Of course, this means either that the computation of coefficients must be done on some other machine with larger registers (not typically available) or that you write a set of extended-precision routines. This, not surprisingly, is the genesis of the BC and DC commands on UNIX, which I certainly did not write for the general public, but for my own use in creating the math library. I am still surprised to find so many people as addicted to the use of these programs as to any habit-forming drug.

I am also surprised that some almost inadvertent choices that I made in the design of DC and BC has resulted in almost complete portability from machine to machine. One of these choices was to use bytes as the fundamental data unit in BC. This seemed even at the time to be a curious choice, but it turns out that any other choice, including the obvious choice of using the machine floating-point registers, would have resulted in serious portability problems.

I would like to make some comments on the current state of mathematical software on UNIX systems as they have evolved since I stopped working on the problem about eight years ago.

One of the most annoying places for inaccuracy to creep into machine arithmetic is in input conversion. If numbers are not accurately converted when read into programs or into a compiler, the whole task of providing accurate answers becomes essentially impossible. Even trying to run tests to find the source of the problem becomes perplexing, since both the argument and the result are suspect. This was the most serious and most time-consuming problem that I encountered back in 1960 and in many versions of UNIX it still persists. In some cases, this inaccuracy is due to ordinary carelessness, but in most cases it depends on an unwise design choice by the designer of the input routines. One might think that if your machine has a 57-bit fraction in its floating point representation (as in the PDP-11 or VAX), then it is not necessary to specify a decimal input number to more than the 17 digits that correspond in accuracy to the 57 bits. It then follows (one might think) that you need not and perhaps ought not even read any digits past the 17th or 18th significant digit of an input number, but can discard them. This is quite wrong, as analysis will show that if this is done, then constants cannot be input that are accurate to the last bit. I and my customers, however, are often interested in the last bit. This problem has become widespread, and, in fact, the majority of complaints that I have received in the last decade about inaccurate math library functions were traced to inaccurate input conversion in the compiler that was used to compile the routines.

Back in the bad old days of the early 60's, I observed that all machines had reasonable mechanisms for detecting floating-point overflow and underflow and divide check. The facilities for reporting these events to the machine-language programmer were also reasonably designed. On the other hand, the operating system for the 7090 and for other contemporary machines did not provide a well-designed mechanism for a FORTRAN (or other high-order) language programmer to do anything reasonable about the occurrence of such an event. I have given this problem a good deal of thought, but I came to no solution, and the provisions for handling arithmetic anomalies in UNIX are barely usable and are in fact little used.

I hoped for a time that the careful design of the exception conditions in the IEEE floating-point standard would cure these problems, but events to date indicate that looked at from the high-order language point of view, the provisions that undoubtedly exist at the machine- language level are not available in useful form. I certainly would appreciate being able to take the numerical data output from a program and feed it into another program. I have yet to encounter a machine that uses the IEEE standard on which this can be done.

I am disappointed at the recent trend to distributing UNIX without source code. That trend, combined with the relative carelessness of software in implementing arithmetic has two effects. The users are stuck with software that does not provide the accuracy that their hardware is capable of, and, in addition, there is very little that they can do about it unless they are willing and able to reimplement the necessary functions from scratch. Another effect is to stifle innovation. I can assure you that it is extremely difficult to write these small but intricate programs without adequate models to work from. You find yourself standing on your predecessors toes rather than on their shoulders, to use the simile of Isaac Newton.

I also notice with some surprise that the collection of math library routines that I decided, in moments of madness in addition to moments of sanity, has not been changed. In all versions of UNIX that I have encountered, not a single member of the library has been added or removed.

To sum up, I have been both a major provider and a major user of numerical software for nearly thirty years, and despite some successes, the principal goals have not been reached. I urge those of you that care about numerical software to rise up and take action.

UNIX V.3 and Beyond

Ian Stewartson

Data Logic Limited
System Software Development Group

ABSTRACT

The object of this paper is to provide an overview of the current state of the UNIX Operating System environment with specific reference to the latest release (V.3) from AT&T. As UNIX has been selected as the basis for a portable operating system by a number of standards bodies, the work being done by these groups is also reviewed. Finally, the paper highlights possible and likely future developments of UNIX that are designed to improve its commercial viability.

1. What Has Gone Before

"Where shall I begin, please your Majesty?" he asked. "Begin at the beginning," said the King, gravely, "and go on till you come to the end: then stop." (Alice in Wonderland)

UNIX has its beginnings back in the middle sixties in the development of the MULTICS operating system – a collaboration between GE (USA), MIT and Bell Labs. The objectives of the MULTICS project were to provide simultaneous interactive computer access to a large community of users, to supply ample computation power and data storage and to allow the users to share their data easily. Although an early version of MULTICS did run, it became clear that it would not meet its goals, and so Bell withdrew from the project.

After Bell Labs withdrew, some members of the Bell team sought to create a computing environment for facilitating programming research and development. Starting with the development of simulation programmes on a GE645 mainframe, this work eventually led to the production of UNIX on a PDP-7.

The requirement for a text processing system within Bell Labs provided the spur to further develop UNIX. These developments included the transfer from the PDP-7 to a PDP-11 in 1971, the development of the C language and the re-write of the kernel in C in 1973. During this time, the number of installations at Bell grew to approximately 25. With the spread of UNIX thorough Bell Labs came the development of a wide variety of utilities and kernel enhancements as each lab evolved UNIX to meet their specific requirements.

Because of a Consent decree signed by AT&T with the US government, AT&T could not market computer products. However, it did provide UNIX to universities and colleges who requested it. To conform with the requirements of the decree, AT&T did not advertise, market or support UNIX. Despite this, its popularity grew, particularly because of the popularity of PDP-11 series, the availability of the sources and the low licence fee for research institutions. By 1977, there were approximately 500 system sites world-wide. Licences for commercial institutions now began to appear along with the first commercial software (an office automation package). This year also saw the first port to a non-PDP machine.

Between 1977 and 1982, Bell combined the various variants of UNIX that then existed into a single commercial system known as System III. This progressed by way of further enhancements to System V which became the first officially supported release in 1983.

1980 was a watershed year for UNIX. Two decisions by AT&T where responsible for this: 1) the introduction of binary sub-licences; and 2) the port of UNIX to a VAX.

1.1. Binary Sub-licences

After the release of the 7th Edition in 1979, AT&T took a major step which led to the appearance of UNIX in the commercial field – the introduction of binary sub-licences. These allowed systems developers to purchase a UNIX source licence, port and develop the system for their particular hardware and market requirements and then sell the complete system in the commercial market. Prior to this, commercial UNIX licences came packaged with a number of unattractive provisions: no warranty; no support; and no maintenance.

Binary sub-licences allowed commercial system developers to port UNIX and sell the binaries to the commercial world with their own warranty and support. It also provided system developers with a relatively easy method of providing a multi-user, multi-tasking operating system on their equipment.

Probably the most successful commercial version was Xenix developed by Microsoft for 16-bit micro-processors and in particular the Intel chip set. Microsoft added a number of enhancements which included: improved and simplified system administration, configuration and set up; improved file system resilience and recovery; power fail recovery; and improved inter-process communications.

These developments and the popularity of Intel´s chip set has led to Xenix representing 70% to 80% of the commercial UNIX licences. Early this year, an agreement with AT&T has resulted in a work programme that will result in the convergence of Xenix 5 (the latest version) with System V for the Intel 80386.

Another major development in the UNIX field has been the appearance of 32-bit micro-processor systems and in particular the Motorola 680x0 series. This has led to the development of UNIX for the high performance workstation market and more recently the commercial market with the delay in the announcement and release of a 32-bit chip from Intel. These workstations, each more powerful than a DEC VAX (the traditional equipment for the scientific market) have also helped the commercial acceptance of UNIX.

1.2. VAX Port

When AT&T released their VAX port in 1979, it was little more than a straight transfer of the PDP-11 code to the VAX. The VAX version (32V) took little notice of the new features that the DEC 32-bit machine had over the old 16-bit PDP-11s. However, given UNIX´s and DEC´s acceptance in the academic world, 32V and VAXs appeared in a number of colleges and universities. The University of California at Berkeley was one of these sites. Berkeley proceeded to develop 32V and convert it to a true VAX version (BSD) which made use of all the new features on the VAX. These included virtual memory, a fast file system and a number of performance enhancements.

The first addition to 32V was virtual memory, demand paging and page replacement. This memory management work convinced Defence Advanced Research Projects Agency (DARPA) to fund Berkeley for the later development of a standard UNIX system for government usage (4 BSD). One of the goals of this project was to provide support for the DARPA Internet networking protocols TCP/IP. This was done in a generalised manner, and it is possible to communicate among diverse network facilities ranging from local networks (Ethernet) to long-haul networks (ARPANET). BSD UNIX rapidly gained acceptance in the VAX community and later with the advent of the 32-bit micro-processor in the workstation market. As a result, BSD represents the third major variant of UNIX and has come to dominate the DEC and workstation markets.

2. AT&T System V.3

"Mr Greenslade! Stop taking those naughty elderly men´s get fit hormones." (The Mighty Wurlitzer)

In 1983, AT&T released their first fully supported version of UNIX – UNIX System V. Initially, this release consisted of little more than some performance enhancements over the earlier version and some new utilities. AT&T continued to develop System V in the following years and releases 1 and 2 appeared. Once again, these releases consisted of performance enhancements and new utilities.

By the time System V.3 appeared in 1986, a number of major changes had taken place. System V.3 is the most radical change in the UNIX kernel in the past 5 years (since 7th Edition). In addition to completely restructuring the kernel and cleaning up utility interfaces, AT&T have added the following new features:

- Remote File Sharing
- Streams
- File System Switch
- Transport Layer Interface
- Internationalisation
- Demand Paging

- Shared Libraries
- File & Record Locking
- Signal Enhancements.

2.1. Remote File Sharing

UNIX has been criticised for its lack of support of networking services. Previously, networking packages have been developed in an ad hoc manner using the traditional character I/O sub-system. Each package defined its own interfaces to the available networking services and its own family of protocols, thus creating a large number of incompatible, inconsistent network protocols and utilities. System V.3 attempts to resolve some of these problems by providing a framework for networking.

Remote File Sharing (RFS) is a software package that allows computers running UNIX V.3 to share resources selectively and transparently across a network. Both files and peripherals (printers, terminals, etc.) can be shared, providing the same functionality on both the host and remote systems. Sufficient security mechanisms are provided to assure local System Administrators of the confidentiality and integrity of their own data.

The main goals of RFS are:

- *Transparent Access.* The standard UNIX interfaces are preserved. The functionality of the UNIX file system under RFS is independent of location
- *Semantics.* The UNIX semantics are preserved. All file types including special devices (printer, terminals) and named pipes can be accessed across the network. Both file and record locking is supported
- *Binary Compatibility.* Existing applications do not require modification or re-compilation to make use of the networked resources
- *Network Independence.* RFS is independent of the underlying network. It operates without modification over a variety of networks that range from LAN´s to large concatenated networks
- *Portability.* Code is machine independent and localised in the kernel
- *Performance.* Minimal network access.

RFS is implemented using Streams and File System Switch (FSS) and a client/server architecture. The Streams mechanism provides Network independence and the FSS mechanism provides transparency. Security is provided via user/group mapping between machines on the network.

During start up, a UNIX system advertises the file systems which can be shared. This information is maintained by primary and secondary name servers. Advertised file systems can be then be *mounted* in a similar manner to local file systems. Additional security features can be used to restrict access to remote file systems.

The *Server* is a kernel process which is not associated with a particular client machine/process whose function is to receive and execute requests from client machines. The requests are processed as if they were initiated from a process on the server machine. On completion of the request, any requested resource is returned along with any error indication. The server is a transaction based process.

Client state information is *recorded* by the server and this allows recovery in case of a machine crash. The recovery mechanism restores the state of the server in the case of client failure and cleans up the client in the case of server failure. On the client side, in addition to cleaning up the kernel information, a user defined daemon is started to perform user level recovery.

The problems which result from networking machines which may have different *time of day*s that will cause an inconsistent view of a file age are resolved using a time delta approach. Any system time-based information is modified by this delta prior to its return to the user application.

2.2. Streams

The nature of the software and hardware communications environments existing in the early 1970s when UNIX was initially developed, influenced the functionality of the character I/O mechanism present in UNIX. However, the emphasis on modularity and performance that is to be found else where in the system does not appear here.

The origina₁ character I/O mechanism, designed for supporting terminals and which processes one character at a time, made the development of software to support a broader range of devices, speeds and protocols difficult. The lack of standard interfaces in UNIX created additional problems when attempting to implement the current generation of protocols with their diverse functionality, layered organisation and various feature options. Attempts to compensate for these problems led to numerous, ad-hoc implementations with protocol drivers inextricably intertwined with the hardware configuration. As a result, software portability, adaptability and re-use have suffered.

In an attempt to resolve these problems, the *STREAMS* facility was developed. It defines standard interfaces for character I/O within the kernel and between the kernel and the user which are simple, open-ended and network architecture independent. Using these interfaces, the user can create, use and dismantle a **stream**. The **stream** is a full-duplex processing and data transfer path between a device driver and a user process. It consists of three parts: a head which handles the interface with the user process; zero or more modules which process the data passing between the head and the end; and an end which is either a device or an internal software driver.

The stream modules, which are kernel resident, can be dynamically selected and interconnected to provide any rational processing sequence. This modularity allows:

- User level programmes that are independent of underlying protocol layers
- Network architectures and higher level protocols that are independent of the lower levels
- Higher level services that can be created by selecting and connecting lower level services and protocols
- Enhanced portability of protocol modules resulting from the well-defined structures and user interfaces.

2.3. File System Switch

File System Switch (FSS) provides for the simultaneous support of different file-system implementations without the application being aware of the difference. It also allows different types of file system to co-exist on the same machine. The main use of this functionality is in RFS; but it is also used to support the 1K file system introduced with System V as well as the older 512 byte file systems, thus making upgrading to System V.3 considerably easier.

The FSS code intervenes between kernel and file system code, isolating different file system implementations. To do this, the generic file system information is separated from file system specific information by dividing the inode structure into 2 parts. Whenever inode specific information is to be processed, the FSS switches the operation to relevant file system implementation. Below that, the standard disk cache functions can be used to transfer blocks.

2.4. Internationalisation

UNIX has progressed in many ways from its original design. However, it is still essentially American with in-built assumptions about character sets, collation sequence, date representation, time zones etc. AT&T have determined the direction UNIX will go in order to internationalise it.

- Removal of dependence on 7-bit US ASCII character set. In System V.3, a subset of the utilities and the kernel have been converted to support 8-bit character sets. The main problem in converting to 8-bit sets is that of sign-extension when converting 8-bit character values to integers
- Removal of System Messages and Text that are hardcoded in English. Messages are no longer hard-coded and are stored separately from the programme At run-time the appropriate message catalogue is accessed depending on the current language. New languages can then be introduced without rebuilding software. This is defined as the future direction by AT&T and no action has been taken in V.3 to implement this
- Expansion of support for non-US habits and conventions. A local customs database provides date/time formats, day/month names, currency formats, radix characters and yes/no strings. Once again, the appropriate database is accessed at run-time. This is defined as future direction by AT&T and no action has been taken in V.3 to implement this.

2.5. Transport Layer Interface

In System V.3, AT&T have taken the first step towards defining networking environment for System V. At present, there are a large number of protocols (OSI – Factory and International; TCP/IP – Scientific and Defence; XNS – Office automation; SNA – Back Office) and communication media (PSS, SDLC, ISDN, CSMA/CD, Token Bus, Token ring, etc.). Each protocol has its own user interface, set of applications and specific media. This has created a number of problems with system connectability leading to user confusion.

AT&T has chosen *STREAMS* as the mechanism by which the problems of the various protocols and media will be resolved to provide consistency; they have also defined a *Transport Layer Interface* which provides a standard user interface to network services.

The *Transport Layer Interface* provides the user with access to standard protocol services in the ISO Transport Service Interface. The Transport Provider Interface specifies the capabilities that must be provided by a *STREAMS* based transport provider and the interface to those capabilities required in order to maintain consistency with the *Transport Layer Interface*. Taken together, they provide the means by which network independent applications can be written. Examples in System V.3 are Remote File Sharing and UUCP (UNIX to UNIX Copy facility).

Future directions defined for standard user interfaces include the support of CASE (ISO level 7) and IBM's LU6.2.

2.6. Demand Paging

System V.3 has introduced demand paging including file mapping into AT&T standard version of UNIX to replace the original segmented architecture. This functionality provides performance improvements for large programme applications as the complete programme no longer requires to be in memory in order to execute. This provides the user with a better start-up response time and allows processes which are larger than the physical memory to be executed. In addition, processes with large amounts of infrequently used code make more efficient use of the system.

2.7. Shared Libraries

Shared Libraries allow a binary application to be dynamically linked to an executable library function at runtime. Significant disk and memory savings can result from the use of shared libraries. Functions in the library are stored once on disk and once in memory where they are shared by all applications using them. In addition to space savings, the effort required to update a library is reduced as only the library need be rebuilt; the new version of a function can then be used automatically by applications accessing that function.

2.8. File & Record Locking

System V.3 provides two types of File and Record locking:

- *Advisory*: This type of lock allows the user to lock/unlock a region of a file. Any other process attempting to lock a locked region will either go to sleep until the region is unlocked or receive an error notification. The locks are advisory in the sense that I/O to a region locked by another process is not inhibited.
- *Mandatory*: This type of lock has the same functionality as advisory locks with the exception that I/O to a region locked by another process is inhibited.

The type of locking is selected on a per file basis depending on the setting of the group access permission.

2.9. Signal Enhancements

The signal processing in System V.3 has been enhanced to resolve some problems with processes missing signals. New signal services have been provided to allow the suspension of signal processing during critical sections of code and the timing windows have been removed to prevent processes missing signals.

3. UNIX and Standards

"The nice thing about standards is that there are so many to choose from." (Anon.)

Portability is an attractive attribute of UNIX. Just about every UNIX supplier has tinkered with the system in some way: tuning up the kernel or adding enhancements. The number of variants is large as can be seen from the list below:

- AT&T *external* releases: V6, V7, System III, System V, System V.2, System V.2.1, System V.2.2, System V.3, ...
- Berkeley: BSD 2.8, 2.9, 4.1, 4.2, 4.3, ..

- IBM: CPIX, PC/IX, VM/IX, Xenix, Series/1 IX, IX/370, AIX, ...
- Others: IS/1, IS/3, IN/ix, Xenix (many variants), Ultrix (many variants), UniPlus, Unity, Venix, HP/UX, ...
- UNIX look-alikes: IDRIS, Coherent, UNOS, CTIX, CTSS-UNIX, ...

Because of the increasing number of variants of UNIX, the need for a standard version became apparent for a number of reasons: users want a standard UNIX so that they have the advantages of a standard operating system (portability, protection of hardware and software investments); vendors want the security of an operating system that will remain constant, adhering to established definitions; and both users and some vendors want to be free from proprietary operating systems – a vendor independent standard. As a result of these requirements, work started on standards in various groupings, all of which are interrelated in some way.

3.1. System V Interface Definitions (SVID)

The purpose of the SVID is to serve as a single reference source for external interface to UNIX System V systems. In the SVID, AT&T specify the complete operating system environment. Its function is to allow the development of applications for UNIX, independent of particular hardware environment. It defines the operating system components for end users and applications. As with all the standards, it is the functionality and not the implementation method which are defined.

AT&T attempted to make conformance to the SVID mandatory in the licencing of System V.3. However, this condition met with strong resistance from the UNIX community (particularly the major manufacturers) and AT&T backed down. It should be noted that conformance is still required for the use of the phrase *derived from AT&T UNIX V.3*.

The SVID consists of three volumes: 1) base system and kernel extensions; 2) utilities (basic, advanced, software development and administration) and terminal interface; and 3) various extensions to volumes 1 & 2.

In order to provide implementation independence, all implementation specific information has been removed and symbolic constants have been introduced where necessary. The SVID provides for various levels of support in terms of the Base system which is mandatory and various extensions to that Base system which are optional. It also defines the method by which a component evolves from one level to another.

Based on the SVID, AT&T have written a System V Validation Suite (SVVS) which checks for compliance with the SVID.

3.2. POSIX and IEEE

Back in 1981, the US UNIX User Group began the drive towards a standard version of UNIX. A standards committee was formed to formulate, adopt, publish and promote a portable system interface based on UNIX. Their proposed standard was adopted and published by the User Group in 1984. It was based on subset of UNIX System III with removal of machine dependent and non-symbolic constants.

When the IEEE P1003 Working Group began preparing the **P**ortable **O**perating **S**ystem for Computer Environment standard, they took the User Group's standard as a first draft. They also adopted the User Group's goal of producing an interface standard rather than a UNIX standard. However, the parentage of the standard is expressed in its name – POSIX. The IEEE Working Group refined and enhanced the User Group's work and included the later developments in the various flavours of UNIX. The other major activity of the Working Group has been to gain wider acceptance for the POSIX standard, initially from ANSI and ultimately from ISO.

The work of the P1003 committee is divided into four sub-committees:

1. *System Interface group.* The goal of this group is to define the external characteristics and facilities of system interface to provide source level portability based on the ANSI X3J11 C standard
2. *Shell and Tools group.* The goal of this group is to define a standard interface and environment for application programmes that require the services of a *shell* command interpreter and a set of command utility programmes or commands. The *System Interface* is not a requirement for this standard
3. *Test Methodology group.* The purpose of this group is to define the standard for test methods for measuring conformance to the POSIX standard. The US National Bureau of Standards has agreed to develop a reference implementation of the conformance test suite which will be placed in the public

domain.

4. *Real Time group.* This sub-committee was fonned from the US User Group Real Time sub-committee in mid 1987. Its function is to explore and evaluate the minimum set of changes and enhancements necessary to allow POSIX to support real time applications. Such changes under consideration include priority scheduling, shared memory, contiguous files and asynchronous event notification.

The work of the System Interface group started in 1984 and resulted in a trial use document which was published in April 1986. The full standard is expected to the released early in 1988. The work of the other two committees only started fairly recently and the final standards are not expected to be ready before Easter 1988.

The work of the POSIX committees has received major support from all major computer corporations and from many government bodies. A number of government bodies have Declared their intention to require POSIX conformance for new systems when it is a full IEEE standard.

After acceptance by IEEE, the standard will probably proceed to ANSI who generally accept IEEE standards without ballot or amendment. Once accepted by ANSI, it is likely that a number of US government bodies will back the standard: National Bureau for Standards, Department of Defence, Federal Information Processing Standards. Beyond ANSI is ISO and in particular Sub-committee 22 of Technical Committee 97. The POSIX committees are currently working with the ISO Sub-committee 22 to review the trial use document with the aim of moving it into the international arena.

3.3. X/OPEN

Early in 1984, ICL and other European manufacturers recognised the difficulties of selling proprietary operating systems into new markets at the low end. Increasingly, systems were being sold on the basis of applications software rather than hardware and the high investment needed in applications software would restrict its ability to penetrate new markets. The lack of a large installed base in those markets was also discouraging third parties from building applications. As a result of discussions ICL had with other European Computer Manufacturers (ECMs), the X/Open group was set up to resolve some of these problems. The X/Open group initially consisted of five major ECMs (Bull, ICL, Siemens, Olivetti and Nixdorf). Since then it has grown to include all seven major ECMs and six manufacturers from the US – DEC, Unisys, Hewlett-Packard, AT&T, NCR and Sun. The major missing manufacturer is IBM.

3.4. X/Open Objectives

> *"To define a complete environment for portable applications, it is also necessary to satisfy the requirement for data management, distributed systems, the use of high level languages and the many other aspects involved in providing a comprehensive applications interface. X/Open intends, therefore, to publish progressively definitions covering these areas."*

The group has a charter to invest business, technical and marketing resources in development of a multivendor Common Applications Environment (CAE) based on de facto and international standards. CAE includes base services (OS and library interfaces), source transfer, the C, Fortran, Pascal and Cobol languages, OS extensions, data management including relational database, commands and utilities, native language system (internationalisation) and interprocess communications. The result of this charter has been the production of a Portability Guide, the first release of which was published in 1985 which was followed by a fully revised and expanded version in 1987.

At present, the group has initiated work in a number of new technical areas which they believe lack a degree of coherence and have a significant future. These areas cover verification procedures, transaction processing, security, graphics (windows), real time and networking.

3.5. Portability Guide

The Portability Guide currently consists of 5 volumes: commands and utilities; system calls and libraries; supplementary definitions, including internationalisation and terminal interfaces, interprocess communications, and source code transfer; definitions for C, Fortran, Pascal and Cobol languages; and data management items.

The guide was originally based on the SVID which has been extended in some areas: some future directions indicated by the SVID have become requirements and the use of symbolic names has been extended. The differences from the SVID are clearly marked. All the SVID base interfaces are maintained as mandatory. There are some optional system calls and library functions. Some of the interfaces are affected by internationalisation and some are still subject to change in the future. The full SVID terminal I/O interface is not supported in the guide because of the requirements of the European market. Other definitions have been affected by standardisation work going on outside the guide.

The first conforming systems are now available and by the autumn of 1987, the then group of eleven members were expected to deliver compliant systems.

The benefits to the user of the guide are independence from a single source of supply, protection of investment in software through increased portability and simplified integration of heterogeneous environments.

In the autumn of 1987, the X/Open group incorporated itself into a independent non-profit making company to *make its independence tangible*, and to encourage greater participation in X/Open activities from independent software vendors and users. The benefits of this later goal have yet to be realised in the light of recent suggestions that the Portability Guide is not being used as it was intended by these independents.

3.6. Convergence

From the above discussion, it would appear that UNIX is being pulled in three difference directions by the various standards groups. However, two major strategy announcements earlier in the summer of 1987 should result in the convergence of these standards:

- The first announcement was the membership of AT&T in the X/Open group with a commitment to conform to the CAE
- The second announcement was the convergence of X/Open with the POSIX initiative when it achieves full-use standard in early in 1988.

These announcements mean that the operating system definition in the SVID will be a subset of the X/Open CAE and that the mandatory parts of POSIX will be a subset of both.

4. To Boldly Go - Beyond V.3

''America was thus clearly top nation, and History came to a .'' (1066 and all that)

In this section, some of the most common/popular areas of UNIX development within the commercial field are examined. These are:

- Transaction Processing
- Networking
- Internationalisation
- User Interface
- Real Time
- MS-DOS/UNIX Convergence
- Security.

4.1. Transaction Processing

One of the major areas for growth in the UNIX market is that of departmental/distributed transaction processing. Such systems will need to operate in large networks in a peer to peer and node to mainframe context. The software infrastructure needed to surround UNIX to support this role is fairly sophisticated. In addition to the provision of a distributed file and database system, several additional layers of logical control are required to provide transparency, integrity and security.

The fundamental purpose of a Transaction Processing (TP) system is to carry out **transactions**. A transaction which is typically performed against a file system or database is known as a **database transaction**. It is performed via the execution of application-specified sequences initiated by a **start transaction** operation and terminated by a **commit**, in the case of successful termination or **rollback** in the case of unsuccessful termination. The commit operation makes all the changes to transaction resources permanent; the rollback operation restores the transaction resources to their state at the time of the start operation – effectively rolling back the changes. Thus the transaction unit is **atomic** – either the transaction is committed and all the changes have been made or the transaction is rolled back and no changes are made.

The transaction processing system provides a framework for the application programmes to effectively and concurrently share the resources support within the system. **Online Transaction Processing** (OLTP) systems are those which are optimised to support interactive applications in which transactions submitted by users at terminals are processed as soon as they are received, with the results being returned to the user in a relatively short time. In addition to interactive applications, **batch applications** can also be supported by an OLTP system and these may be initiated by an interactive application or some system event such as time of day.

Distributed transactions are those which are processed via the co-operative execution of multiple application programmes. When OLTP systems are linked, they can share their resources with one another, thereby facilitating the implementation of distributed transaction processing applications.

Application programmes operate upon local resources (terminals, databases, queues, etc) and co-operate with other programmes by exchanging data and co-ordinating their state. The initiate/terminate operations need to be applied to all the co-operating programmes involved in the transaction. In order that these operations may be applied across a distributed transaction, the local resources that are updated by the co-operating programmes during the execution of the distributed transaction must be protected at the local level by similar initiate/terminate operation. Only protected resources can be reliably managed with the distributed transaction.

The application programmes which co-operate during the execution of the transaction can be located remotely from the terminal which originated the transaction. In this event, the programmes, each of which is managed by its own local OLTP system, communicate together to form a **Distributed TP Environment**.

The resources managed within this environment can be classified as follows:

- *User File/Database resources.* The user file and database management systems are concerned with the security and consistency of the physical and logical structure of the user´s data
- *Transaction Processing resources.* The system is responsible for the scheduling and management of application programmes supported by the OLTP, and the conversations between them
- *Network resources.* The individual OLTP systems which make up the distributed TP environment must be connected by a network
- *Peripheral resources.* The OLTP is responsible for the allocation of peripherals, for terminal emulation and all I/O functions. Form management and the security employed by these devices should also be managed with the OLTP system.

In order to support transaction processing environments, there are a number of areas in which UNIX kernel modifications may be required. These areas are listed below. The first two are be regarded as essential for effective implementation. The rest provide ease of implementation and operational efficiency.

- *Synchronous Input/Output.* This ensures that on completion of an I/O operation, the data has been transferred to the physical medium
- *Concurrent Input/Output.* This allows one process to support multiple terminals, changing the normal UNIX architecture of one process per terminal
- *High Performance Files.* These provide faster access to data because of the high demand that OLTP systems normally place on disk sub-systems
- *Process In-Memory Locking.* This is a performance enhancement for high-priority processes to reduce swapping or paging
- *Improved Scheduling Strategy.* This provides for process bias and pre-emptive scheduling as another aid to performance improvement
- *Concurrent Read Performance.* At present, UNIX restricts the number of reads outstanding on a given file to one even if the data for the other requests is in the cache. Increasing the number of outstanding reads will improve the performance
- *Multi-Volume Files.* Users with large data volumes often require more filestore that can be handled by one physical disk volume. The provision of multi-volume files makes the support of large data volumes easier.

The provision of these features under UNIX is eminently feasible. At present, X/Open´s consultants in this field, Data Logic, have produced a white paper describing the transaction processing model proposal. It provides a open architecture that should be able to incorporate disparate vendor products. It also presents in detail the protocol boundaries associated with each functional level, within different application architectures. This paper is currently under review by X/Open members and other interested parties.

4.2. Networking

In today´s data processing world, system and product diversity are common place as well as the links between these systems. However, one of the major problems is to provide access to data on other machines without having to write special software to access the remote data or to transfer the data between machines. The main goal of developing a distributed file system is to provide the illusion of a single file system whilest distributing access to this data across a network.

To provide this illusion, it is necessary to have:

- Transparent Access
- Reliability
- Concurrent Access Control
- Security Access Control.

AT&T´s solution, RFS, has already been discussed earlier in this paper. There are, however, two other products on the market which attempt to provide heterogeneous distributed file access, Sun´s Network File System (NFS) and The Newcastle Connection.

Of the two, NFS is of more interest since it has gained wide acceptance with the UNIX community and is competing directly with RFS. The salient idea behind the Newcastle Connection is the concept of a *super-root* directory which exists above the normal root directory of all systems on the network. Complex network topologies can be generated using a hierarchy of super-roots. However, this concept has not been widely adopted.

NFS is similar to RFS, particularly in the use of servers and clients. The areas where the design goals differ from RFS are:

- *Machine and Operating System Independence.* The protocols used should be independent of UNIX so that an NFS server can supply files to many different types of client. The protocols should be simple enough that they can be implemented on low end machines like PCs
- *Crash Recovery.* When clients mount remote file systems from many different servers, it is very important that clients be able to recover easily from server crash.

The other major different between NFS and RFS is the starting base for development. NFS is based on BSD UNIX, whereas RFS is based on System V.3.

One of the problems Sun faced when developing NFS was distinguishing between local and remote files. RFS resolves this problem using the *File System Switch* (FSS). Sun altered the kernel to support a *Virtual File System* (VFS) which replaces all the parts of the kernel involved in the access of files. This compares with FSS which only replaces those parts which process inodes. If a file is located on a remote system, the VFS uses the NFS protocol to access and manipulate the file. This protocol is a set of primitives that define the operations which can be performed on a distributed file system. In contrast to RFS, NFS uses a stateless protocol. No state information is maintained making recovery very easy. The nature of the implementation is such that a client cannot distinguish between a slow server and one that has just performed recovery.

NFS uses a remote procedure call to execute a primitive on the remote machine. Both these procedure calls and the protocols are implemented in terms of a lower level protocol known as the External Data Representation which defines byte ordering, size and data type alignment in a machine independent manner. Thus data is transferred in a common format.

The main strengths of NFS are error recovery, system independence and availability. NFS has been available under licence from Sun for the past three years. During that time, it has built up a wide acceptance in both the commercial and academic fields and has become the de facto standard.

NFS is currently available under UNIX, VMS and MS-DOS whereas RFS is currently restricted to AT&T´s 3B series. However, since RFS is part of System V.3, it is likely to be available from all manufacturers who support V.3.

The main weaknesses of NFS are:

- *UNIX Semantics.* NFS fails to maintain all UNIX operating system semantics for remote files. Append mode and file/record locking are not supported. Although the locking problem appears to have been resolved recently via the use of a lock server to handle file/record locking. Remote files can be deleted whilst in use by another user. However, it should be remembered that NFS attempts to support non-UNIX file systems so a complete mapping should not be expected

- *Security Control.* Although the actions for the super-user are limited across the network, there are no similar restrictions on user. A user id on one system maps directly onto the same id on another system, potentially providing access to restricted files. The facility, *Yellow Pages*, provides the ability to define one password file across the network. Since this is a voluntary mechanism, security can be compromised by the least secure system
- *Time skew.* No attempt is made to resolve time-skew problems across machines.

At present, Sun are working to improve NFS in the areas of diskless systems, time-skew, remote file locking, support of other file systems, performance enhancements, improved security and improved portability of the server.

A recently announced collaboration between AT&T and Sun may result in the merger of the two systems such that a system running NFS can be attached to an RFS network and visa versa.

4.3. Internationalisation

The objective of the internationalisation effort is to provide a system-wide framework to enable application software to be adapted for use in any country or local environment. This implies the support of character set specifications, classification tables, message catalogue, functions to manipulate characters and messages, and a local customs database. In addition to converting applications to support internationalisation, work is also required on the development of the message catalogues and library functions.

The X/Open specification is based on Hewlett Packard's Native Language Support (NLS) software was originally developed for their MPE operating system. The design goals of this system are: integrity of data, support of multiple extended character sets, local language user interface, local conventions support, one version of software for all languages, capability of extension for new language/culture/code set and minimal kernel level changes. In addition to the work already done or defined by the SVID, NLS also covers:

- Extended character set support. At present, UNIX supports 7 bit ASCII externally and 8-bit quantities internally. Many Roman characters and all middle/far Eastern characters are not supported. Full 8 bit external support will provide additional characters. X/Open specify the 8-bit IS8859/1 code set which supports all the European languages and is compatible with ASCII. The X/Open specification for NLS allows for multiple 8-bit characters sets to be supported simultaneously on the same machine
- Language Announcement. This function allows the user or programme to switch between languages as and when necessary
- Character Identification. Character class, conversion (up/down shift) and collation tables for each language are required to determine character attributes. Collation functions are defined to handle 1 to 2, 2 to 1 and *don't care* mappings.

The NLS has been implemented by Hewlett-Packard and is now available to other manufacturers under licence.

The future directions specified by X/Open include:

- The extension of number of 8-bit clean programmes. At present, only 22 programmes are specified as 8-bit clean
- The resolution of the regular expression *class* metacharacter problem. This metacharacter allows the matching of a range of characters: for example an alphabetic match is expressed as *[a-zA-Z]*. This is correct in English but not in Swedish (for example)
- Convergence with ANSI X3J11 C standard
- 16 and 32 bit code sets for Far eastern languages. This includes the support of mixed code set and character size (8 and 16 bit).

4.4. User Interface

Windows systems are becoming an indispensable part of UNIX and until recently, X-Windows (from MIT) and NeWS (from Sun) were vying for acceptance and supremacy.

The development of Window systems was started in the early 70's by Xerox at their Palo Alto Research Centre (PARC) and this work has been the basis for all subsequent Window systems. PARC were responsible for the development of most if not all the basic ideas that are now mandatory parts of a window system: overlapping windows; icons; menus; and scrollbars. In 1980's, a number of proprietary systems began to appear. However, these systems had serious limitations, particularly in the areas of portability and networking.

At the same time, research projects in US at CMU and MIT were developing systems which attempted to solve these problems. In 1984, ANSI also began to look at requirements of window systems which resulted in the formation of X3H3.6 Technical Sub-committee to develop a standard for Window Management Systems.

The main issues for Window systems are:

- *Stability*. No commercially available window system is more than five years old and no system has remained unchanged for more than one year. This is of particular importance since it is through the windows that an end user interacts with their applications
- *Portability*. Window applications are difficult to build and require a non-trivial amount of time to lean how to use. Therefore, a consistent programmer interface is important. X-Windows has been ported to a wide variety of machines from PCs to supercomputers and most of the major workstations. However, NeWS is based on PostScript which drives a variety of printers making picture printout easy and has been ported to a number of PCs. ANSI's purpose is to provide a standard environment for developing window applications.
- *Toolkit*. Panels, buttons, scrollbars and other tools have become an essential part of most modern window-system applications. Most proprietary systems offer them as standard. The latest version of X-Windows (X.11) contains a composite toolkit based on those developed by a number of organisations for earlier versions. A NeWS toolkit supporting Sun's SunView toolkit is expected in the near future.
  The toolkit does not lock the system into a given user interface. Functions are hidden behind interactive items (icons etc.) which can be made to look like almost anything. Thus, the integration of a standard system into existing user interfaces can be done without too many problems
- *Graphics*. The emergence of window systems does not preclude already established graphics standards from fitting into the window standard. It has not yet been established how a graphics standard such as GKS or PHIGS would fit in with the window environment. But both NeWS and X-Windows provide hooks to allow additional commands to be added to the base set, allowing GKS/PHIGS calls to be supported in the same manner as the standard windowing functions
- *Performance*. At present there are no independent comparisons of X-Windows and NeWS. In theory, NeWS should be faster in complex situations as it communications via a language rather that procedure calls. Unlike previous ANSI standards, performance is an integral part of the window system.

Both NeWS and X-Windows have a client-server architecture which allows access to applications from remote intelligent terminals. The server is specific to the capabilities of a given hardware device, but has a common interface on the network side. This interface communications with the client application and informs the client about the capabilities of the hardware. Window calls issued by the client are translated into the desired display or the best approximation to it.

The two key differences between the two systems are a) the attitude to pixels; and b) the interface to the server. Under X-Windows, the fundamental concept is the pixel. The client manipulates these pixels via a procedural interface. Whereas, under NeWS, the fundamental concept is the path (a set of curves defined in the coordinate system and rendered on the device by stroking or filling). The client manipulates the path via a programming language which is transferred to the device. The use of a programming language has a significant effect on the communications bandwidth needed to operate a display remotely in cases of fast interactive feedback or large numbers of repetitive operations.

During 1987, the development direction for windows became more clear. In January of that year, eleven leading manufacturers (including DEC, HP and Apollo) endorsed the X-Windows approach. This was followed by an announcement by the X/Open group that it was looking to standardise on X-Windows. In September, the latest version of X-Windows (V.11) was shipped by MIT. Late in the year, the ANSI sub-committee on Window Management agreed to a request to investigate X-Windows as a basis for a future standard. The committee is currently developing reference models and documentation for X-Windows as a standard as well as determining the relationships between X-Windows and the various graphics standards (GKS, PHIGS, etc). Other developments at the end of the year included a proposal from DEC and Sun on the method for including of 3-D graphics in X-Windows and the formation of an industrially sponsored consortium to continue the development of X-Windows which is expected to include 98% of all workstation manufacturers. As a result, X-Windows has almost become the de facto standard with little support outside Sun for NeWS.

However in the long term, it is likely that NeWS or something similar will supersede X-Windows, particularly as device resolution increases from the current 50 to 90 dpi to 200 or more dpi.

4.5. Real Time

A Real Time O/S (RTOS) differs from a general purpose system in being deterministic – i.e. it responds to external events with a set time limit. UNIX was designed as a *general purpose, interactive time-sharing* operating system and lacks many RTOS features.

Two questions arise as a result: 1) what functions are features are required for an RTOS; and 2) how can UNIX be extended or re-designed to provide these capabilities? The later gives rise to one further question: is it worth the effort to provide Real Time on UNIX? This arises because standard UNIX incurs overheads to support resource sharing. Resource sharing is inherently non-deterministic. As a result, UNIX may not be able to deliver real-time response times.

At present, a number of large manufacturing companies (Hughes Aircraft, Chrylser, General Motors, IBM, HP, AT&T, DEC, et al.) are looking at real time extensions to UNIX. The RTOS requirements under consideration by the various UNIX standards groups are:

- Event and IPC mechanism
- Shared Memory
- Memory control (resident locking, access to physical I/O)
- Priority scheduling and pre-emption concerns
- Asynchronous I/O facilities
- Synchronous files (write through files) and physical I/O completion
- Reliable signal mechanism
- High Performance file access
- Privilege allocation
- High precision clock
- Metrics and terminology.

These shortcomings in UNIX are not insurmountable. However, the primary question to be answered is whether trade-offs involved in resolving these shortcoming make UNIX a practical base to start from. It should be noted in this context that General Motors and Chrysler have both developed versions of real-time UNIX controlling manufacturing plant.

Three major methods have currently been used to provide real-time facilities under UNIX:

- *Real-Time Kernel Extensions.* Under this method real-time extensions are added to the kernel in addition to a number of other modifications to the kernel to give the user more effective control. A typical example is UNOS developed by Charles Rivers with extensions for priority interrupts and scheduling, contiguous files and enhanced performance and interprocess communications. Vanilla UNIX only allows pre-emption and context switching to take place at the start or end of a kernel service. This can cause interrupt latencies in the range of 1 second or more. Hewlett-Packard have developed a real-time kernel by adding additional pre-emption points in addition to other real-time features.
  The modification/extension of the kernel is compatible with the current standards because the real-time services are transparent to them and run underneath.
- *Real-Time Kernel.* Under this method, the kernel is completely re-written to provide a UNIX interface on top of a real-time kernel. This technique provides application portability by fooling UNIX applications which think that they are running on UNIX and have no idea of what really exists behind the kernel interface. Examples of this technique include Alcyon's Regulus and AT&T's Mert.
  In general, these systems use a two layer approach to implementing the real-time functionality. The lowest level provides the real-time services which are the most difficult to implement in the standard kernel. These include pre-emptive and priority scheduling mechanisms, asynchronous I/O, synchronisation and contiguous files. On top of this low level kernel (but still part of the kernel) sits a UNIX supervisor which interfaces with the low level services. The supervisor provides the UNIX environment and functionality to applications which run on top of the supervisor in user mode.

- *Distributed Executive.* An alternative architecture integrates UNIX with a dedicated real time executive via distributed processing techniques. A host processor supports an environment for developing applications and for execution of traditional applications and parts of real time applications. The satellite processors run dedicated real-time executives. The satellite processors may or may not be UNIX systems as they do not require compilers and debuggers and may not even require file systems and schedulers. These satellite systems may have customised versions of UNIX with features specific to the task they have to perform.
  For example, an individual real-time process may be specific to the sensor or device it is collecting data from or controlling, but the rest of the application can be running under UNIX.

4.6. MS-DOS/UNIX Convergence

Despite their version of Xenix, Microsoft´s major success has been in the single user PC market with their MS-DOS operating system and its associated products. The success of the PC and MS-DOS has resulted in a large number of commercial packages being available for these systems, ranging from word processing to database, spreadsheets to graphics. Since MS-DOS first appeared on the Intel 8088 based IBM PC, the PC market has progressed first to full 16-bit support and now to full 32-bit support with the advent of Intel´s latest chip, the 80386 which has a raw CPU speed of 3 to 4 MIPS. However, MS-DOS itself has changed little since its original version and is viewed in some parts of the industry as a *toy* operating system. Although support for large disks, international versions and UNIX file semantics have been added, the operating system remains restricted in the original 8088 environment - 640K bytes of memory and single task/single user. It has not evolved to take advantage of the features of the later generation of Intel chips. These include: virtual memory, linear address space, configurable protection and task control functions.

In particular, the new Intel 80386 is probably the most ludicrously overpowered engine ever to be crammed under the bonnet of a single user machine, given that its raw CPU speed approachs that of a CDC6600. Current predictions suggest that it should be able to support a 20 user UNIX system with full memory protection. Intel themselves have aimed the 80386 at the multiuser market currently led by Motorola with the 680x0 series.

A multi-user version of MS-DOS should have pre-empted much of the current interest in UNIX. But Microsoft has not produced this. Even the new OS/2 cannot truly be said to be multi-user/task and is unlikely to be ready and stable for some time. Most of the current interest in the 80386 is UNIX based which can take full advantage of the chip´s paging and large address space (64T byte). In this area, Intel are likely to profit from their commitment to compatibility that has been much criticised in the past.

Until the advent of the 80386, there were three methods of providing MS-DOS under UNIX: software emulation of Intel´s 8086; a hardware co-processor; or using a PC/AT under Xenix and MS-DOS. All of these suffered from the problems of cost, system overheads, badly behaved programmes (with the first two methods) which directly addressed the hardware by-passing MS-DOS, and single user access to MS-DOS (with the latter two method). In addition, under Xenix, the user has to re-boot the system as although Xenix can read and write MS-DOS disks, MS-DOS applications have to be run under MS-DOS. In addition, UNIX workstation (which are mainly 680x0 based) manufacturers have recognised the importance of providing access to UNIX´s technical and MS-DOS´s commercial software on the same box.

The design of the 80386 allows it to run in one of several modes on a per-task basis and to switch rapidly between these modes. Of the available modes, two are of importance to UNIX: protected mode; and virtual 8086 mode. Virtual 8086 mode is a sub-set of protected mode which provides the protection and demand paging of protected mode and 8086 programme execution in the same manner as an 8086. This allows the chip to run UNIX in protected mode and MS-DOS in virtual 8086 mode at the same time, switching between modes on a per-task basis which gives multi-user access to MS-DOS.

With the arrival of the 80386, two major products have appeared which allow multiple versions of MS-DOS to run under UNIX: VP/ix from Interactive Systems; and VM/Merge from Locus. The functionality of the 80386 allows the host operating system to detect and process interrupts and direct access to the hardware allowing even badly behaved applications to run. A user under UNIX can start up an MS-DOS application o MS-DOS itself and when working under MS-DOS can initiate a UNIX application as if they were running under UNIX directly. The development of Window systems allows the user to have MS-DOS and UNIX applications active concurrently on the same terminal. Even badly behaved MS-DOS programmes are supported by these products, something that cannot be said of OS/2 under some circumstances.

MS-DOS files are integrated with the UNIX file system to avoid the need to partition the disks or add filename prefixes. An I/O re-director translates the files into the appropriate format when reading from or writing to the disk. Thus UNIX and MS-DOS files are interchangeable. Each MS-DOS application *sees* 1M byte of accessible memory; however, the programmes are paged in and out of 64k byte pages along with the rest of the system.

4.7. Security

Until recently, highly secure systems have existed almost exclusively in defence establishments. However, since deregulation of the City and the rise in computer related crime, interest in security has surged in the commercial sector. Much of this interest is based on work already done in the defence field. Although most defence establishments have their own criteria for defining and selecting secure systems, only the US Department of Defence (DoD) has published its criteria and classification. As a result, much of the discussion on secure systems is based on the system security classification described in the DoD's *Trusted Systems Evaluation Criteria (TCSEC)*, also called the *Orange Book.*

In the TCSEC, the collection of system elements responsible for implementing the security policy is called the Trusted Computing Base (TCB). Only the TCB is analysed to assess its security provision. In the case of UNIX, this means as a minimum the kernel and the utilities **init**, **getty**, and **login**. Because security depends on the hardware as well as the software, both are evaluated and the complete system receives a classification. The TCSEC bases its classification on four criteria:

1. *Security Policy.* This criterion specifies how the TCB must control access between subjects and objects. It area covers: discretionary access control; object re-use; label integrity including exportation of label information to multilevel devices and human readable output, subject sensitive labels and device labels; and mandatory access control.
2. *Accountability.* This criterion defines the rules for how the TCB must keep a record of security related events. These first two criteria specify the visible security features. This field defines two areas: a) identification and authentication including trusted paths; and b) audit trails.
3. *Assurance.* This criterion specifies the attributes of the TCB's design and/or structure that ensure that the security features are implemented securely. It covers two areas: a) operational and b) life-cycle assurance. Operational assurance covers system architecture, system integrity, convert channel analysis, trusted facility management and trusted recovery Life-cycle assurance covers security training, design specification and verification, configuration management and trusted distribution.
4. *Documentation.* This criterion defines the documentation that must be provided in order to operate a secure system. This field covers four areas: a) security features user's guide; b) trusted facility manual; c) test documentation; and d) design documentation.

TCSEC defines four divisions of security (A to D) within which there can be a number of sub-divisions or classes (numbered from 1 – lowest). Each class has a specific set of criteria (based on the general criteria mentioned above) which fall into four groups. The divisions in order of increasing security are:

D *Minimal protection* (1 class). It denotes systems that have been evaluated but but fail to meet the requirements for higher levels

C *Discretionary protection* (2 classes).
The system provides users with controlled access to data on an individual basis. It also requires some auditing of actions performed by a user in the area of system security.
UTX32/S (Gould) and VMS4.2 (DEC) are classified as level C2 (highest in division C).

B *Mandatory protection* (3 classes). The system provides qualitatively stronger protection than level C. It requires mandatory access control that enforces a systemwide classification of data according to the system security policy. Enter *positive* vetting. New requirements are added over and above level C.
MULTICS is classified as level B2.

A *Verified protection* (1 class). The principle addition is the requirement under Design Specification and Verification for a *Formal Top Level Specification* that describes the TCB in a formal language (usually a form of first order predicate calculus). This must be shown to be consistent with the *Formal Model of Security Policy* (also described in the same formal language) which is a requirement for the highest division B class and with the source code for the TCB.
SCOMP(Honeywell) is classified as level A1.

At the lower end of the classification (C1 to B2), the emphasis is on additional security features. At the higher end (B2 to A1), the emphasis switches to assurance. However, nothing is discarded in moving to higher security levels as the TCSEC is structured such that the requirements of a particular level apply to that level and all higher levels.

As well as recognising that security is a new *selling feature*, manufacturers have been spurred on by a change in US law which came into force in late 1987 which requires that all computers purchased by the government must be classified at level C2 and that the top 25% need to be level B1 or above.

Vanilla UNIX is almost at level C1. Most of work required to raise UNIX to full C1 level is in the area of security documentation. Other features which need to be addressed include: login; password usage, aging, generation and allocation; obtaining super-user privileges; secure distribution channels; and writable shared resources.

For UNIX to move to high levels, in addition to the work required to introduce mandatory access control, work is required in the areas of Accountability (vetting of user access and audit trails) and Documentation (provision of adequate security documentation). Level B3 and above will require a complete re-write of the kernel in addition to any new functionality.

At present, work is currently under way by X/Open, the UNIX user group and POSIX committees to define a standard more secure UNIX.

The appearance of networked systems has raised more problems for the security of a system which the *Orange Book* does not address, in particular the security of messages on the network. There are two projects currently under way, one in the UK and one in the US which are attempting to address networking. Both projects are based on the current *Orange Book*. The UK DoTI is expected to publish its networking criteria by 1988. In the US, a semi-classified document known informally as the *Brown Book* which addresses the problems of networked systems is currently available on limited circulation. The US DoD is also currently evaluating the security criteria for Database system, but no date is available for the issue of these criteria.

5. The Future

"What are we going to do now?" (Spike Milligan)

UNIX has grown and matured over the past twenty years from a small in-house project to one of the major vendor-independent operating systems. Given the current developments outlined earlier in this paper, UNIX is certainly the most mature of the vendor-independent systems and is as feature-rich as many proprietary systems.

The three major changes that are likely to place in the UNIX field in the next five years are a) the rise of POSIX standard; b) the development of the secure UNIX systems; and c) the increased use of LAN based networked systems linking UNIX and MS-DOS & OS/2 systems. During this time, the POSIX standard will become the main starting base for the development of UNIX-like systems, replacing AT&T UNIX. This will allow vendors to keep their proprietary operating systems with any competitive advantage this may give and as well as provide their users with access to a broader range of software applications. These include the developments of newer operating systems which are POSIX conformant, yet provide features or scope not available under standard UNIX. Examples include: OS/9 and TRON (currently under development in Japan).

These developments are being driven by the commercial requirements of, in particular, the US government and these requirements place a number of constraints on the technical development by forcing it down a number of paths – minimal kernel changes, compatibility with other versions etc. It is questionable whether this is the right approach in view of the requirements of transaction processing, security and real-time processing.

The networking of UNIX and MS-DOS systems is a very recent development, given the large number of personnel computers in the commercial field. A number of manufacturers are developing integrated office products which allow these PCs to share information and resources with a central host. Indeed, X/Open have set up a working group to look into the question of networking PCs and UNIX systems via LANs.

The other area which is attracting considerable attention is that of RISC architecture. A large number of RISC machines are now available from major manufacturers such as IBM and HP, through others like Sun, to the smaller manufacturers such as Immos and Acorn. RISC machines are also expected from the traditional micro suppliers Intel and Motorola. The common feature of all these machines it that they run UNIX under one version or another. Part of the recently announced relationship between AT&T and Sun is to develop an Applications Binary Interface standard to allow binary portability across machines. Much of this work is based on Sun's new RISC chip (SPARC).

There are a number of uncertainties on the immediate horizon which arise from the failure of the major corporate developers of UNIX to clearly indicate their future directions. In particular, the relationship between AT&T and Sun and the position of IBM with respect to UNIX give cause for concern. Whilst the

AT&T/Sun relationship will bring benefits to both the protagonists and UNIX, the relationship is causing concern in the industry over benefits Sun may gain over its competitors. The benefits to UNIX include the re-writing of the kernel to merge System V.3 and SunOS which is 4.2 BSD based. IBM's position with respect to UNIX is more complex. UNIX is now available in one form or another across the complete IBM range with the recent announcements of AIX on the 9370 and the PS/2. However, questions arise over IBM's commitment to UNIX in the commercial market as they have not made their position clear with positive actions instead of just words. It does look, however, that *even* IBM realise that UNIX has and will continue to have an important role in certain markets: workstations, graphics, government.

6. Further Reading

''And there's more were that came from.'' (The Famous Eccles)

AT&T. "System V.3 Documentation"

Bach M.J., "The Design of the UNIX Operating System", Prentice-Hall Software Series.

Department of Defence. "Trusted Computer System Evaluation Criteria: DOD 5200.28-STD"

Fraim, L. "SCOMP: A Solution to the Multilevel Security Problem", *IEEE Computer Volume 16 Number 7*, 1983 pp 26-34.

Haviland K. and Salama B., "UNIX System Programming", published by Addison-Wesley Publishing Company.

Hopgood, F.R.A. et al (eds), "Methodology of Window-Managers", published by North-Holland.

IEEE Computer Society, "IEEE Trial-Use Standard: Portable Operating System for Computer Environments", published by John Wiley & Sons, Inc.

Presotto, D.L., and Ritchie, D.M. "Interprocess Communication in Eighth Edition UNIX System", *Usenix Conference Proceedings*, June 1985

Quarterman, Silberschatz and Peterson. "4.2 BSD and 4.3 BSD as Examples of UNIX Systems", *Computing Surveys, Volume 17 Number 4*, 1985 pp 379-418.

Rifkin, Forbes, Hamilton, et al. "RFS Architectural Overview", *Usenix Conference Proceedings*, June 1986

Ritchie, D.M. "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal*, 63(8) (October 1984)

Sandberg, Russel, Goldberg, et al. "Design and Implementation of the Sun Network File System", *Usenix Conference Proceedings*, June 1985

Scheifler R.W. and Gettys, "The X Window System", *Transactions in Graphics* Number 63, 1986.

Sun Microsystems, Inc., "NeWS Preliminary Technical Overview", *Sun Microsystems*, 1986.

Teitelman, W., "A Tour Through Cedar", *IEEE Software, Volume 1, Number 2*, 1984 pp 44-73.

Weinberger, P.J. "The Version 8 Network File System", *Usenix Conference Proceedings*, June 1984

Whitemore, J., et al. "Design for MULTICS Security Enhancements, ESD-TR-74-176", *Honeywell Information Systems, Inc.*, 1974

X/Open, "X/Open OnLine Transaction Processing Reference Model", published by X/Open

X/Open, "X/Open Portability Guide, Volumes 1 to 5", published by Elsevier Science Publishers B.V.

General Purpose Transaction Support Features for the UNIX Operating System

S. G. Marcie
R. L. Holt

NCR Corporation
E&M Columbia
W. Columbia, South Carolina 29169

ABSTRACT

This paper describes the features of NCR's General Purpose Transaction Facility (GPTF), an extension to NCR's implement-ation of UNIX System V for the TOWER supermicrocomputer. Timer signals with millisecond resolution are presented. Performance of process synchronization and interprocess communication is improved via a set of semaphore primitives which executes in the user program environment and operates on structures which exist in standard UNIX System V shared memory. A scheduler is described which reduces process switching latency and provides process scheduling among both realtime and timesharing priority classes.

Additionally, a mechanism is provided to lock a process in memory so that it is immune to paging. Scheduling latency is reduced through voluntary preemption within the kernel. A novel disk I/O scheduler provides the ability to schedule disk requests according to process priority, seek distance, or some configurable combination of both parameters.

User access to the transaction processing facilities is provided via a set of system calls and shell commands. A user friendly interface is provided to allow a superuser to control such access.

1. Introduction

The UNIX Operating System provides interprocess communication, timer signals, and a process scheduler which are well-suited for time-sharing applications. Applications which control automatic teller machines, cash registers, and other response oriented devices, however, require more efficient and predictable processing of the "transactions" which they service.

An example of a transaction oriented application is one which controls point of sale terminals handling sales of goods to customers in a department store. Such an application could be part of a system which also performs other functions relating to the store's daily operation such as maintaining inventory, ordering, and price data bases; generating sales reports; and updating payroll information.

A typical terminal control system might use a standard relational data base, such as UNIFY, to maintain information pertaining to goods for sale in the store. While the point of sale terminals will issue queries to the data base for price information, other processes might also be accessing the data base. An inventory control application, for example, could be updating inventory levels as items are sold, a data collector might be retrieving retail and wholesale prices for end-of-day report generation, and a data capture process could be recording all events occurring in the system. Each request for access to the data base might be directed by a request router to an appropriate queue. One queue would be maintained for each type of data base request. In many UNIX systems, these request queues would be implemented using System V message queues. When there is one or more requests on the request queues, a semaphore could be set indicating to a request server that there is an access pending.

A point of sale terminal control system such as this is transaction oriented. The time required to service a transaction, such as the price lookup request from a terminal, directly translates into wait time for the customer. The lower bound on this wait time is defined by such things as the processor speed and disk driver throughput. The upper bound, however, depends upon the number of other processes in the system and, due to the scheduler implementation in many UNIX systems, this could exceed acceptable limits for transaction oriented processes.

Transaction oriented applications require predictable response times. GPTF provides a modified process scheduler, a unique disk I/O scheduler, and improved system features which provide an environment for development of transaction oriented applications.

2. Process Management

UNIX was designed to provide a "fair" time-sharing environment in which all user processes would receive an equal opportunity at using system resources. The scheduler and memory management schemes which achieve this prevent UNIX from being an effective environment for transaction oriented applications. The difference between the time the CPU stops executing one process and the time it begins executing the next is referred to as switching latency. Switching latencies in typical UNIX systems tend to vary so much that process run time is no longer predictable to within reasonable limits. While system resources such as processing power and disk throughput are fixed, the way these resources are allocated is dependent on the system scheduler. Transaction oriented applications require a scheduler and system features which can reduce switching latencies and order disk accesses to provide bounded, and thus predictable, response times.

2.1. Residency

A typical UNIX system utilizes paging for memory management. If available memory runs low, the kernel will free up memory by writing pages to secondary storage. Any time a reference is made to a page which is not resident in memory, a page fault results and the kernel retrieves the associated page from secondary storage. The length of time required to page in enough memory to begin execution of a process is one component of switching latency. In many UNIX systems, the system call *plock()* provides the ability to lock a process in memory, thereby avoiding the delays involved in paging memory to disk. There is, however, no way to specify a limit on the amount of memory which must be reserved for non-resident processes. This means that enough processes could invoke the *plock()* system call to lock all available memory, preventing initiation of additional processes.

GPTF provides a system call to make a process resident which provides a major advantage over *plock()*. When a process is made resident via the GPTF system call, the pages associated with that process are touched. This means that if they are paged out at the time the system call is executed, those pages will be paged and locked into memory at that time. If the process expands, the new data is also made resident. The key advantage of GPTF residency is the ability to specify, via a parameter in the system configuration file, a lower limit on the amount of memory which must be reserved for non-resident processes. This helps to ensure that there will be enough memory available to begin execution of other user processes.

2.2. Priorities and Process Scheduling

In all UNIX systems, every process has an associated priority. For many systems, this priority is a numerical value from 0 to 127 where a lower numerical value corresponds to a "better" priority. Processes with priority in the range 0 through 39 are kernel processes, while those with priority in the range 40 through 127 are user processes. A process which is suspended waiting for a resource which is controlled by the kernel will receive a kernel priority when it resumes. Processes are subject to aging algorithms, which means that as a process runs, its priority is degraded. The limitation inherent in such a scheme is that there is no way to distinguish a process, such as a transaction oriented process, as being significantly "more important" than any other process. Some UNIX systems permit the superuser to issue the *nice()* system call with a negative argument to assign a better priority to a process. This priority will still be aged, however, eventually becoming degraded beyond its original value. Transaction oriented applications require a system which allows the specification of the importance of one user process relative to another.

When the scheduler selects the next process for execution by the CPU in typical UNIX systems, it searches the run queue for the process with the highest priority. In such an implementation, the time required to search the run queue becomes a component of switching latency. This search time varies widely depending upon the number and priority of processes awaiting execution. Transaction oriented applications require a scheduler which reduces to a predictable value the time required to search the run queue.

The GPTF scheduler maintains a traditional UNIX time-sharing environment while simultaneously providing a responsive transaction oriented environment. This is achieved by breaking user processes into two classes: real-time processes and time-sharing processes. The priority of real-time processes is fixed over the range 40 to 59, where 40 is the "best" user priority in the system. A real-time process will run to completion unless either it becomes blocked waiting for a kernel resource or another process of better priority becomes scheduled. The time-sharing process group has a priority range from 60 to 127 and these processes are subject to the typical UNIX time-sharing strategy of CPU allocation. Time-sharing processes are further broken down into two classes: time-sharing processes with fixed priorities (fixed time-sharing processes), and time-sharing processes with priorities which are aged (dynamic time-sharing processes).

Because the priority of real-time processes and fixed time-sharing processes are static, the *nice()* system call has no effect on these types of processes. For dynamic time-sharing processes, *nice()* can be used to adjust a process priority. For instance, the superuser can use *nice()* with a negative argument to give a time-sharing process a priority in the real-time priority range (40..59). If, for example, a dynamic time-sharing process of priority 65 is the subject of a "nice --15" call, this process will then have a dynamic time-sharing priority of 50 and will compete with real-time processes for system resources until its priority is degraded back into the 60 to 127 range.

The scheduler changes are designed to favor real-time processes and to provide more predictable switching times. While many UNIX systems maintain only one run queue, the GPTF scheduler maintains four run queues, one for each of the following four types of processes:

1 real-time priority processes in kernel mode

2 real-time priority processes in user mode

3 time-sharing/fixed priority processes in kernel mode

4 time-sharing/fixed priority processes in user mode

Inserting a ready process into the run queue in typical UNIX systems involves merely placing that process at the head of the run queue. In GPTF, however, the queues for the first three process types listed above are ordered based on process priority. When a process is placed on one of these run queues, it is inserted in that queue relative to its priority, with processes of better priority placed toward the head of the queue. This reduces latency by making selection of the next ready process more efficient. The queue for the fourth process type listed above is not ordered because the necessary overhead would be too large (recall that dynamic time-sharing processes may have their priorities recalculated while they are on the run queue).

When choosing a process to run, the order in which the run queues are searched depends upon whether one or more real-time processes are waiting for resources in the kernel. Instances occur when time-sharing processes using kernel resources must be allowed to finish execution to free resources for waiting real-time processes. If no real-time processes are waiting for resources in the kernel, the queues are searched 1-2-3-4. If one or more real-time processes are waiting for resources in the kernel, the queues are searched 1-3-2-4. Also, among several real-time processes, the one with the best priority will be selected but the system will not preempt between real-time processes having the same priority because this would be a waste of CPU time.

2.3. Process Switching

The scheduler provided in typical UNIX systems operates on a time-slicing principle. This means that the scheduler allocates the CPU to a process for a specific, pre-determined time interval (called a quantum), preempts the process when the time quantum is exceeded, and reschedules the process for subsequent re-execution by placing the process back on the run queue. When the process is rescheduled, a new priority is calculated based upon how long the process has been running in order to ensure that all processes get equal treatment in using the CPU and other system resources. Such an implementation adds two components to switching latency: the length of time required to recalculate the priority when a process is rescheduled, and the time quantum for which the CPU is allocated to a process. This last element becomes a factor because the scheduler in many systems has only one preemption point. Because this preemption point resides at the point where a process stops executing or becomes blocked, the system will only check the run queue for a better priority process after either the current process has exceeded its time quantum, the current process has completed, or the current process has blocked waiting for a resource. Transaction oriented applications require a system which can reduce the latency due to changing priorities and can provide more timely switching when a better priority process becomes scheduled [1].

In addition to the new scheduling algorithm, GPTF implements a new CPU management scheme which further reduces the switching latency for high priority processes. Recall that the priorities of real-time processes and fixed time-sharing processes are static. This means that when such processes change from kernel to user mode, time is not spent calculating a new priority. Moreover, when any process changes from kernel to user mode, a preemption check is made. If a better priority process is on one of the run queues, the current process is preempted and execution of the better priority process begins. Preemption points have been placed in lengthy kernel operations, such as when a pathname is resolved to its corresponding inode in *nami()* or when a process executes a *fork()* system call. If a better priority process is found on any run queue at the check point, the current process is preempted. This helps to prevent the CPU from being inaccessible to a better priority process for any significant amount of time.

3. I/O Scheduling

Disk accesses in many disk driver implementations are prioritized according to the seek distance with respect to the current direction of head motion. Consider the following example. A process generating a monthly report of sales figures may be requesting a price lookup from a data base. This data may physically reside on the disk two cylinders "in front of" where the head is currently positioned. A better priority process controlling a point of sale terminal may be requesting a price lookup from a data base to generate the cost of a sale. If the data for this second request is four cylinders in front of the head, the first request will be honored first, even though the second request is, in a response time sense, more important.

For a transaction oriented process, it is sometimes desirable to forsake the strict seek distance ordering of disk requests. GPTF provides the ability to schedule disk I/O according to the requesting process's priority, the relative seek distance required to service the request, or some combination of the two. The value of a disk priority parameter in the system configuration file determines how disk I/O requests are scheduled. A value of 0 specifies that all disk I/O is to be considered to be of equal priority, and therefore all requests are sorted by seek distance only. A value other than 0 specifies the number of process priority queues to be created for disk requests. For example, if 4 is specified, four queues will be created: one for disk requests issued by processes with priorities in the range 0 through 31, one for requests by processes in the range 32 through 63, one for requests by processes in the range 64 through 95, and one for requests by processes in the range 96 through 127. Within each queue, requests are sorted by seek distance. Requests in the first queue are honored first, followed by requests in the second, etc.

4. Event Synchronization

Some environments require tight control over closely cooperating processes. Two or more applications, for example, may be sharing system resources such as memory buffers or data bases and must interact frequently, communicating synchronously or asynchronously. System V interprocess communication (IPC) facilities were designed to satisfy this need, but they suffer from poor performance and a baroque programmer interface [2]. GPTF addresses these System V shortcomings by providing a new set of timers which have millisecond resolution, and by supplementing IPC facilities with faster semaphores and an alternate message queuing scheme. Implemented in shared memory, the GPTF semaphores and message queues provide more efficient synchronization, enhanced performance, and greater flexibility.

4.1. High Resolution Timers

UNIX provides timers via the *sleep()* and *alarm()* system calls. While these have resolution measured in seconds, transaction oriented applications typically require timers with resolution in milliseconds.

The timer services provided by GPTF are a supplement to UNIX System V timers. The GPTF timers may be single-shot where the timeout occurs only once after the timer is set; or periodic where the timer is made active again each time a timeout occurs. The GPTF timers may be further specified to be maskable or non-maskable. If a timeout occurs on a maskable timer while the process is in a timer-disable state, the process's timeout routine is not called until the process enters the timer-enable state. If a timeout occurs on a non-maskable timer, the process's timeout routine will be called independent of the timer enable/disable state.

The GPTF timers allow transaction oriented applications to request a timeout with an accuracy down to one "tick" of the clock driver. The number of GPTF timer structures in the system is regulated by a parameter in the system configuration file. While the *alarm()* system call will set at most one alarm request per calling process, the number of GPTF timers which may be set by a process is limited only by the number of free timer structures in the system. Although multiple GPTF timers may be set by a process, any of those timers may be cancelled individually.

While the GPTF timers provide greatly enhanced timer services, such features are useful to transaction oriented applications only if they do not exact a heavy performance cost. Processing of GPTF timers induces no overhead on other system services, and the overhead involved in updating the timers is negligible and independent of the number of outstanding timers. The overhead involved in processing a timeout signal is similar to the overhead in processing signals in typical UNIX systems.

4.2. Shared Memory Semaphores

Because semaphores in typical UNIX systems reside in and are maintained by the kernel, any operation on the semaphore requires entry into the kernel. This kernel entry is very time consuming and can be too costly for transaction oriented applications.

GPTF semaphores are counting integers residing in memory which is shared between the processes using them. A process attempts to lock a shared resource via a call to a C runtime library routine. This routine decrements the semaphore counter and if, after the decrement, the semaphore counter is non-negative, the routine returns without entering the kernel. If, however, the semaphore counter is negative, the kernel is entered to queue the process on a chain of processes waiting for that resource (redundant code in the kernel serves to ensure the integrity of the semaphore structure). Thus, processes which lock a shared resource do not require entry into the kernel if the resource is available. The result is that GPTF semaphores are up to 12 times faster than semaphores in typical UNIX systems in this non-blocking case†. Because the non-blocking case is the most common for many applications, use of GPTF semaphores results in a significant performance improvement over typical UNIX semaphores.

4.3. Shared Memory Message Queues

The System V IPC message queuing scheme is implemented as a linked list of message queue headers. The message queue structure pointed to by each header contains an integer indicating the message type and an array of characters containing the message. While this implementation is suitable for some needs, it is not perfect for every application. Transaction oriented applications require a message queuing scheme which is flexible enough to be worked into that application and efficient enough to not severely impact performance.

GPTF provides the ability to tailor a message queuing scheme to the individual needs of an application. GPTF message queues reside in memory which is shared between the processes using them and they utilize GPTF shared memory semaphores to manage certain variables. Messages take the form of an integer address of a message structure. This structure is designed by the application designer and may take the form of a character array, an array of buffers, another integer pointer, or any other form limited only by the imagination of the application designer. The result is that GPTF message queues are extremely flexible and very fast, especially in the common, non-blocking case where only one process is performing an operation on the message queue at a time.

Another feature of the GPTF message queues is that much of the queue management is handled by a C library interface. GPTF library routines lock and unlock the semaphores which control access to the queues and increment and decrement the indices of messages in the queues. In this way, GPTF provides an efficient message queuing scheme without burdening the user with management tasks.

5. Real-Time Attributes

If *plock()* is used to make a process resident in many UNIX systems, a subsequent *fork()* will create a child process which is not resident. In GPTF, however, it is possible to specify that transaction oriented features such as residency and priority are to be inherited by child processes. Thus, for example, if a process with this inheritance privilege is made resident in memory and that process does a subsequent *fork()*, its child process will also be resident in memory.

Residency in memory and the priority at which a process is specified to run are examples of real-time attributes. GPTF allows real-time attributes to be bestowed upon applications via shell commands as well as through C library calls. Providing shell commands which bestow real-time attributes enhances portability by allowing an application to use transaction oriented features without requiring any code changes to the application itself. This means, for instance, that a transaction oriented application such as the point of sale terminal control system described earlier can use an off-the-shelf data base manager such as UNIFY to maintain store information. Providing C library calls which bestow real-time attributes allows the programmer to code his or her application to take advantage of GPTF's transaction oriented features without having to use special shell commands.

† Based upon a comparison of UNIX System V semaphores and GPTF semaphores on a TOWER 32/600. Test results represent 1000 operations on a resource whose access is controlled by a semaphore. Test system was run with a characteristic system load.

6. Control of Access to Real-Time Functions

The features provided by GPTF can have a profound impact on the performance of a system. As a result, user access to such features must always be carefully monitored and controlled. GPTF provides the superuser with a user-friendly mechanism to strictly regulate the use of GPTF system calls and shell commands.

The superuser may always utilize GPTF functions. For non-superusers, however, GPTF maintains a permissions file containing the user and/or group id's of processes allowed to use GPTF functions. If this file is not present, only the superuser has access to GPTF functions. If this file is present, then any attempt by a non-superuser process to bestow real-time attributes upon any process will result in a scan of this file for the user and group id of the process issuing the call. If such an entry is found and the function requested is within the bounds specified for that id, the action is performed. A shell command permits the superuser to add and to remove id's to and from this file.

7. Future Directions

There are a number of areas which may provide opportunity for increased performance and functionality for transaction oriented applications. Modifications to the file system to support contiguous files would enhance performance and synchronous writes, closes, etc. would provide significant improvement in data integrity for some transaction oriented applications.

Additional places may be identified in the kernel to place preemption checks, which would further reduce the switching latency for high priority processes.

User access to I/O devices, possibly through a shared memory create/attach syntax, and a mechanism to attach user processes to hardware interrupts would provide enhanced functionality for real-time process control applications.

The Institute for Electrical and Electronics Engineers (IEEE), as part of its efforts to define a POSIX standard, has formed the P1003.4 real-time subcommittee. P1003.4 has currently identified eleven areas which should be addressed in a real-time extension to the POSIX standard. GPTF currently addresses nine of these and the other two are addressed by the future directions described above. NCR is a member of 1003.4 and will support the real-time extensions to POSIX and the real-time extensions AT&T is planning for SVID [3].

8. Acknowledgements

We would like to express our appreciation to James L. Browning, Manager of Advanced Systems Development at NCR – E & M Columbia, for his input, and to H. L. Rogers, Manager of TOWER Systems Development at NCR – E & M Columbia, for his encouragement to undertake this paper.

We also wish to express our personal thanks to Sabrina Vansant for the time she spent typesetting this paper.

References

1. Wendy B. Rauch-Hindin, "UNIX Overcomes Its Real-Time Limitations", *UNIX/World*, **5**(*11*), *pp. 64-78, November 1987.*
2. Jim M. Barton, "Can UNIX Take the Plunge?", *UNIX Review*, **5**(10), pp. 59-67, October 1987.
3. Unauthored, "AT&T's Real-Time UNIX", *UNIX/World*, **5**(11) page 68, November 1987.

A toolkit for software configuration management

Axel Mahler, Andreas Lampen

Technische Universität Berlin

ABSTRACT

For almost ten years, *Make* has been a most important tool for development and maintenance of software systems. Its general usefulness and the simple formalism of the *Makefile* made Make one of the most popular UNIX tools. However, with the increased upcoming of software production environments, there is a growing awareness for the matter of *software configuration management* which unveiled a number of shortcomings of Make. Particularly the lack of support for version control and project organization imposed a hard limit on the suitability of Make for more complex development and maintenance applications.

Recently, several programs have been developed to tackle some of the problems not sufficiently solved by Make. **shape**, the system described in this paper, integrates a sophisticated version control system with a significantly improved Make functionality, while retaining full upward compatibility with Makefiles. **shape**'s procedure of identifying appropriate component versions that together form a meaningful system configuration, may be completely controlled by user-supplied *configuration selection rules.* Selection rules are placed in the *Shapefile*, **shape**'s counterpart to the Makefile.

The **shape** system consists of commands for version control and the shape program itself. It is implemented on top of the *Attribute File System* (AFS) interface. The AFS is an abstraction from an underlying data storage facility, such as the UNIX filesystem. The AFS allows to attach any number of attributes to document instances (e.g. one particular version) and to retrieve them by specifying a set of desired attributes rather than giving just a (path–) name. This approach gives an application transparent access to all instances of a document without the need to know anything about their representation. So, it is also possible to employ different data storage facilities, as for instance dedicated software engineering databases.

The project organization scheme of **shape** provides support for small (one man), medium, and large projects (multiple programmers/workstation network).

1. Background

For almost ten years, *Make* [6] has been a most important tool for development and maintenance of software systems. Its general usefulness and the simple formalism of the *Makefile* made Make one of UNIX' most popular tools. However, with the increased upcoming of software production environments, there is a growing awareness for the matter of *software configuration management* which unveiled a number of shortcomings of Make. Particularly the lack of support for version control and project organization imposed a hard limit on the suitability of Make in more complex development and maintenance applications.

The notion *Configuration management* has been introduced by US military and government institutions for a set of management techniques dealing with the complexity (and costs) of very large development and maintenance projects. Triggered off by the surge of interest in programming environments during the last few years, the term *software configuration management* (SCM) made its way from the management domain towards the software engineering and development domain. Configuration management has been tried‡ to define as "the process of identifying the *configuration items* in a *system*, controlling the release and change of these items throughout the system life cycle, recording and reporting the status of

‡ well, if you consider an ANSI standard a *try*

configuration items and change requests, and verifying the completeness and *correctness* of configuration items. It is a discipline applying technical and administrative direction and surveillance to (a) identify the functional and physical characteristics of a *configuration item*, (b) control changes to those characteristics, and (c) record and report change processing and *implementation* status" [9]. Major subtopics included in configuration management are *change control*, *configuration identification*, *configuration control*, *configuration status accounting*, and *configuration audit*.

In the UNIX domain, Make and source control systems as SCCS [13] or RCS [15] are in widespread use as configuration management tools. When designing a new configuration management toolkit for UNIX, one has to have a very close look at what is already there. In the categories given above, the areas addressed by these tools are *configuration identification* and *change control*, the mainly technical aspects of SCM (in contrast to the other, more *management* related disciplines). The configuration management toolkit described in this paper stays in the technical area and attempts to solve some of the problems not sufficiently covered by existing tools, while at the same time laying the groundwork for further developments that implement support for *managing* software projects. The toolkit consists of a dedicated version control system, and **shape**, a significantly enhanced Make program. **shape** has full access to the version control system, and allows the user to specify configuration rules, to control the selection process for component versions during identification, build or rebuild of system configurations. For the time being, we choose to be upward compatible with Make and thus to retain its concept of openness and versatility. Besides, this helps to make a new – potentially complex – tool easy-to-learn and easy-to-use for the large developer community who is familiar with Make.

Since integration of version control was a major objective for our system, we had to face the need for a document identification scheme that goes beyond the usual way of specifying name and type of a document. As a consequence, we began to design an *attributed filesystem* (AFS) introducing a much more generalized scheme for document identification. The AFS comprises concepts for version control, support of variants or status models for example. Furthermore, it helps to abstract from the particular underlying data storage system in such a way that it makes no difference whether it is an ordinary filesystem or a dedicated database system.

In the following section we make a serious effort to use Make and RCS effectively for configuration management purposes and give an impression of some typical SCM related problems that are not (or at least very hard) adequately to control with Make and existing source control systems. Section 3 takes a closer look at how things are improved by **shape**. We explain the new concepts of **shape** and how it works internally. In section 4 we conclude the discussion and and outline the prospects for future work. Samples for a Makefile and a Shapefile that support the discussed activities can be found in appendix A and B respectively.

The paper assumes that you are familiar with the concepts of Make and the Makefile.

2. Doing it with Make

Although the full power of Make's basic concept takes effect only in the UNIX environment, the program has been ported to or reimplemented on a considerable number of systems. Without Make, UNIX wouldn't be what it is today. However, in the last years a number of attempts have been made to improve the program in a number of ways, complete reimplementations have been done, and quite a lot of criticism has (respectfully, though) been expressed, of what is considered weaknesses of Make [2,3,7,8,10,14,16]. Often, the discontent is related to some kind of specialized new task that Make is put to work on, which it wasn't originally designed to perform. So, this kind of criticism can also be interpreted as some compliment for a very flexible tool.

In [16] it is argued that one of Make's more serious drawbacks lies in the *lack of standards* and *differences between versions*. Makefiles are frequently written in an ad-hoc fashion rather than carefully designed, as they should be. An ambivalent issue about Make is that one tends to use it without having to think about it. This situation is supplemented by a lack of education. There are only few tutorials and guides that explain how to make the right use of Make. Even experienced Make-users happen to be unaware of some of Make's features and abilities.

Despite proven deficiencies, there are only few absolute no-no's with Make. An experienced and sufficiently stubborn practitioner will (almost) always be able to invent some workaround that serves her particular need. This results from a consequent concept of openness and extensibility. The combination of Make with tools as *sh, awk, sed, grep, cpp* ... makes Make really a devil of a fellow. Nobody said that all the power is easy to use, though.

In the subsequent discussion we present quite a hard try to (well, *kind of*) integrate Make with RCS, one of the finest available source control systems, intending to create a tool environment suitable for basic software configuration management tasks.

We're addressing three common application scenarios that relate to basic SCM tasks:

1) *cooperating developers:* a number of programmers independently working on different parts of the same targeted system.
2) *system integrator:* a functional role that gathers the work results of cooperating developers and creates *official* configurations (releases). The integrator has *read-only* access to all sources.
3) *maintenance:* backing up formerly saved system configurations for the sake of bug fixing or incorporating customer specific modifications while development goes on at head the system components' main lines of descent.

2.1. A Sample Project Setup

For the first scenario we assume that every programmer has a dedicated directory where the workfiles (we call them *busy versions*) of those components, she is in charge of, are kept. During component development these busy-versions undergo frequent changes. In order to test a work result, the programmer needs to build a complete system configuration from these *privately modified* components and *all remaining* components, possibly controlled (owned, reserved, locked) by other programmers. To avoid interference by temporarily inconsistent components, taken care of by other programmers, it is necessary to maintain consistent (or *milestone*) versions of these components and make them available. Making use of RCS, it's quite simple to organize mutual access to saved, (hopefully) consistent versions of all system components. RCS allows to store document revisions in archive files residing in a special subdirectory, RCS. To circumvent confusion with lots of different pathnames (of all the programmers' work directories) and to provide shareability of the Makefile, it is a good idea to create one single directory as repository for all the RCS archive-files, and let all members of a project have a *symbolic link* (named RCS) to that directory. Unfortunately, this works only with the BSD flavor of the operating system.

To arrange Make and RCS, a couple of new transformation rules, handling dependencies of sources from source archives must be added to the Makefile. A sample Makefile that does the job described here can be found in appendix A. These rules in conjunction with the setting of special macro *VPATH* which extends the scope wherein filenames (in this particular case their *suffixes*) are looked up, cause all .o files which's source is not present in the current directory to be generated from the corresponding, latest saved .c-source in the archive file. Temporarily retrieved sources are deleted immediately after the compilation is finished. The resulting object files, however, are kept and only regenerated when the corresponding source archive is touched, i.e. a new source version becomes available.

An important point in this approach is that cooperating developers have to trust each other in that new module versions are made available with *care*. The described scenario works quite well in preventing programmers from accessing components that are under immediate construction. This does not imply however that *functional changes* or *interface modifications* are properly advertised. In fact, herein lies a much more severe problem, because programming languages like C don't provide detection of these cases. Usage of *lint* would be advised here but for obvious reasons, routine use of lint is – to say the least – unpopular. Some mechanism for either accepting/rejecting of newly submitted versions or incremental testing with associated status control would be helpful.

Eventually needed include files must remain checked out, because Make doesn't allow to express that a derived object depends on (possibly multiple) sources stored in an archive. This might be considered harmless in the context of C but causes considerable overhead when applied to programming languages as Modula-2 or Ada, where each compiled object depends on at least two sources.

Another problem for cooperating developers is to keep their Makefiles consistent. Each programmer needs her own copy of the overall Makefile which should be *identical* for all programmers. Changes to Makefiles, e.g. introduction of new system components, new dependencies or different compiler options, should be published as soon as possible in order to maintain a common level of information. However, most of the changes made by a programmer will be very small, and therefore it is unnecessary to force her to edit – and maybe damage – the entire Makefile, a possibly huge, complex and extremely sensitive piece of information. For cooperative projects, it would be highly desirable to have such things as *distributed* or *modular* Makefiles providing localization of information while at the same time maintaining consistency with the whole of the project.

2.2. Making Releases

The creation of releases is the *system integrator's* job. Releases are system configurations, intended to be passed to the outside world or representing internal milestones. Releasing a product includes taking the (at least moral) responsibility for its functional integrity and to be prepared to react on maintenance requests. Establishing a release baseline means to *synchronize* the development stages of all components that are part of it and creating a *checkpoint* where all independent lines of development intersect and are bound together (i.e. make sure that all program modules work and cooperate properly, and all the documentation is up to date). We're talking testing and reading here, things that require a consciousness of responsibility, and mostly have to be carried out manually. The release process can be fairly complex: work results of programmers and writers might have to be reviewed or evaluated against specifications, unsatisfying results have to be rejected, corrected, evaluated again a.s.o. There is a complex relationship between software configuration management activities such as version control, version status accounting, and configuration auditing and systematic software quality assurance. An in-depth discussion of this matter can be found in [4].

To fix a release requires precise identification of the product configuration. For the sake of maintainability any given release should be exactly reproduceable. This comprises looking up all the source document versions and rebuilding any derived product under the same prerequisites as has been done originally. Although it is impossible to do this a 100% perfect, this can be accomplished to a certain degree by logging all characteristics of a configuration in a configuration identification document (*CID*). To meet minimal requirements, a CID should contain the following information:

- a complete list of the system components with version identification
- the version of the Makefile
- the current date, and the identification of the programmer in charge for the configuration
- identification of all tools and their versions that were involved in building the configuration.

Doing this with Make and RCS is not the easiest thing to do. When dealing with these requirements, it becomes obvious that the functionality of RCS and Make should be truly integrated rather than artistically woven. RCS provides for marking document revisions with state attributes (e.g. "released") and furthermore allows to associate unique symbolic names (e.g. "release2.1") with a revision. Such a symbolic name would typically be shared by all component revisions that are part of a given configuration. The problem is, that RCS is concerned with individual documents rather than (complete) sets of related documents and offers no way to make use of the information stored in the Makefile. Also, the question comes up, how to define the version number of a configuration, i.e. a set of programs, each composed from a number of modules, all with different revision numbers. It would be nice if RCS operations as `ci` could be applied to entire configurations, i.e. *targets* in the Makefile.

Make on the other side, having no idea what release building is all about, has its problems assuring that all components are at least saved and properly marked. Flexible handling of *variants*, be they implemented as conditional compiles, separate files hidden in subdirectories, or variant branches in RCS archives is also a nightmare with Make. When subsequently building different variants of a system, Make either doesn't detect that some files have to be recompiled due to change of compile-flags, or it recompiles all sources from scratch because it is unable to figure out if a particular module is actually *affected* by such a change.

In the sample Makefile from appendix A, when making the target "release", it is made sure, that all components are saved and systematically marked† before the programs are built and the CID is generated. To understand how the complete mechanism works in detail is left as an exercise to the reader.

2.3. Maintaining releases

The third scenario comprises the functions involved when modifications on releases shall be done. These are typically bug fixing, or customer specific tailoring of the released system.

The *maintenance engineer* begins her work with recovering all source versions belonging to the release in question. In order to avoid collisions with the current development versions, this has to be done in a directory especially created for that purpose. This directory should have an RCS subdirectory holding *copies* of the system master source archives. With the two added rules for handling RCS archives, Make is able to check out the appropriate source file versions for a given release name.

† marking accounts for unnecessarily compiling the complete system, because the archives are touched.

During reconstruction of the system it becomes evident, whether the *system integrator* has done a good job. If not all source versions belonging to the release are properly marked, the maintenance engineer is really in trouble. In this case, reconstructing of a formerly consistent configuration will be extremely time consuming. Obviously, a reliable release rebuilding mechanism is essential for the maintainers work.

The process of maintaining an old release causes forking of a new line of development on the basis of obsolete component versions. Depending on the complexity of the change request, a maintenance phase might last quite a while and involve a number of developers. The workspace for the maintenance process, however, should be separated from the main development area to avoid mutual interference. Following the described approach *without copying* the master source archives would cause unnecessary check out operations and recompilations in the main development area, each time a new revision is appended to the maintenance branch (RCS file is touched). Even worse, recompilations with the latest *main stream* revision would occur in the maintenance area, each time a new revision is checked in on the main line of development.

Besides the need for physical separation of the maintenance and development archives (resulting in organizational and diskspace overhead), the check out rule in the Makefile (`.c,v.c:`) has to be rewritten in a way that the latest revisions from the *maintenance branch* are checked out instead of the latest revision on the main line of descent, which is RCS' default. Although it is *possible* to write such rules, this would further increase the complexity of the Makefile while decreasing the performance of the Make process. Creating completely new maintenance archives with the components of the maintained release as initial revisions, would result in inconsistent numbering between maintenance and development versions. A later reincorporation of maintenance branches into the master archives would be impossible.

Although this gives an idea of what can be done with Make‡, we believe that it also becomes obvious why the *do-it-with-Make* approach comes to its limits here. Makefiles of the given kind become extremely hard to write and maintain, are error-prone while the implemented function is still unsatisfying, and react very fragile to any kind of exception. While the first described scenario is still comparatively well supported by the presented approach, the introduction of more complex SCM activities puts a heavy load on Make and an even heavier burden on the Makefile writer.

For professional software production, SCM should be applied on a routine basis. Taking this into consideration, the described scenarios are everything but exotic. Things *have* to be much easier to use, more systematic, and much more robust.

3. How things can be improved

One of the most frequently denounced drawbacks of Make is its inability to react effectively on changes in the transformation environment, e.g. if a compile flag is changed. In discussing this matter, it has often been argued, that Make's restricted understanding of dependency as *time* dependency is the reason for this unsatisfying behavior. However, a second – closer – look at the problem leads to the conclusion that this is a mere *symptom* for the real problem, lying in the inability of the UNIX filesystem to handle more file attributes than those stored in the inode and the directory.

For the realization of **shape**, we tried to overcome the limitations imposed by the UNIX filesystem by creating the *attribute filesystem* interface (AFS) that provides an extended view of documents to application programs. This view comprises the concept of *document histories*††, as well as the possibility of tagging any number of *user-defined attributes* to document versions or complete document histories. Interpretation and use of these attributes is left to the application that uses them. Each *document instance* (i.e. one single version) is understood as a complex of *content data* and a set of *associated attributes*. Document attributes have the general form *name=value*, where *name* represents the attribute name, and *value* a possibly empty attribute value. There are a number of implicitly defined *standard attributes* for every document instance. Some of the standard attributes (e.g. name, size, owner, or protection attributes) are inherited from the UNIX filesystem, others (e.g. revision number or state) are AFS specific.

With AFS, documents are retrieved by specifying an *attribute pattern*. An AFS retrieve operation results in a – possibly empty – set of *document keys*, each of which representing a unique document instance that matches the specified attribute pattern. For the identification of documents, all attributes are considered equally suited. For instance, it is possible to retrieve all documents with an attribute *name=xyzzy*, or all documents with the attributes *author=andy* and *state=published*. The attributes that together form the

‡ I forgot to mention that some minor modifications had to be made in `rlog` (→ `"newrlog"`) and `ident`. `newrlog -y` returns nothing but the latest revision number of the given file.

†† represented as sequences of changes (deltas) [12].

version id guaranteeing the unique identification of one document instance, are under strict control of the AFS. These attributes are *document name*, *document type*, *generation number*, *revision number* and the *variant name*. Generation numbers are used to indicate major development steps that are common to all components of a configuration or subconfiguration. The revision number serves as an update sequence number for individual components within a generation. Generation number and revision number together are often referred to as *version number* (e.g. 3.0, 18.49).

The data presented and accessed by means of the AFS is stored in a data storage system such as the UNIX filesystem or some database. In the current implementation, the AFS abstracts from a set of *archive files* used to host contents and attributes of stored objects. Every regular file can be viewed as an AFS document, even if it has never been touched by an AFS application. In this case, it will be treated as a busy version without AFS specific or user-defined attributes. If such a file is checked into the version control system for instance, a source archive file will be created, and the missing standard attributes will be supplemented in a meaningful manner.

The fundamental difference between Make and **shape** is that Make is naive about versions of objects and **shape** is not. Make assumes that any object it deals with is a UNIX-file and therefore has only one version — the *current*. When evaluating dependencies, Make looks just at *name* and *modification time* of files. This is as good as Make can be on the basis of the UNIX filesystem. The fact that auxiliary tools like RCS had to be introduced in order to provide for multiple revisions of documents without creating name conflicts is another symptom, pointing at the same deficiency of the filesystem. RCS supplements a limited number of attributes such as revision number or state to the set of standard file attributes. Most of RCS' functionality is controlled by these attributes. However, there is no way to use these attributes to extend Make's idea of dependency, and integration of both tools has the character of a *'hack'*.

shape has an understanding of source versions and possibly multiple instances† of *derived objects* that exist at the same time (e.g. objects compiled from different versions of the same module). When producing a target, **shape** associates a number of *derivation attributes* with the resulting derived objects that *sufficiently describe* the current transformation. To describe a transformation *sufficiently* means to record all prerequisites that are needed to reproduce an identical copy of the derived object. Currently, these attributes are the *version ids of all source-objects* and the *flag definitions* that were in effect for the transformation.

While Make bases its decision whether to fire a transformation on the build-in *time-dependency-relation* between the target and its dependents, **shape** checks the target's derivation attributes against the attributes that a new transformation would generate. This causes **shape** to recompile a system if for instance compile flags have been changed, or other but the default (e.g. *newest*) source versions shall be used (for example in case any of them doesn't work as expected and couldn't be integrated). Also, recompilations can be avoided if an instance of a derived object can be found that already has the attributes that would be assigned to a newly produced one. This becomes interesting when more than just one instance of a derived object is kept and a different, previously saved (therefore immutable) source version shall be used. The **shape** toolkit offers support for multiple instances of derived objects stored in a cache-fashion administered *derived object pool*. The concept of derived object pools is borrowed from DSEE [1], a highly sophisticated software engineering environment running under Apollo's AEGIS operating system.

3.1. Elements of the toolkit

Besides facilities for keeping multiple instances of source– and derived objects, the **shape** toolkit provides programs for

- *basic version control*, such as `save` for creating inalterable versions, `retrv` to replace the current busy version by some formerly saved version, `vl`, `vinfo`, and `vcat` to browse information about document histories and view particular versions.
- *project interaction* that are used to organize and control the document submission process. `resrv` (developer) attempts to reserve the update privilege for a document history, `sbmit` (developer) submits a work result to be included into the next release of a system, `accpt` (system integrator) accepts a subitted document version and gives it an official, publically accessible state, and `rject` (system integrator) denies a submitted work result the official state and sends a corresponding message to the submitting programmer.

† it should be noted that only source-objects are subject to version control and bear *version ids*. Derived objects do not have version attributes. Multiple instances of them are solely kept for the sake of efficiency, i.e. to prevent recompilations.

– *building configurations*, namely the program `shape` which builds a system configuration from a *Shapefile* or a *Makefile*.

All programs of the toolkit are implemented on top of the AFS interface and share the same basic concept of document identification. The version control commands can be applied to entire configurations or subconfigurations, because they use the information stored in the Shape– or Makefile (actually, some of the version control programs are *links* to `shape`). In the remaining part of this article we will fully concentrate on the way configurations are built by **shape**.

3.2. The Shapefile

As it is the case with Make, everything that **shape** may do is controlled by a description file, the *Shapefile*. The **shape** program is upward compatible with Make and thus understands Makefiles, which are taken as system description, if no Shapefile can be found in the current execution context (which *can* be more than just the working directory if a project context is active). **shape** behaves identical to Make if a conventional Makefile is interpreted. Significant improvements with respect to Make are the introduction of *selection rules* and *variant definitions* into the description file to control the interaction with the attribute filesystem. Selection rules and variant definitions are expressed differently from the Makefile syntax. The formalism for the definition of default transformation rules has also been extended.

Selection rules control the selection process for component versions during identification, build or rebuild of system configurations. Variant definitions help to deal with the complexity of variant administration.

The Shapefile consists of four main components:

- system description
- selection rules
- transformation rules and
- variant definitions.

The syntax for the system description part is the same as for Makefiles. The remaining Shapefile sections have a slightly different syntax from Makefiles and must be preceded and ended by special comment sequences (`#% RULE-SECTION, #% VARIANT-SECTION, #% END-RULE-SECTION` a.s.o.).

3.3. Configuration Selection Rules

A selection rule is a named sequence of *alternatives*, separated by semicolons, which form a logical OR expression. Each alternative consists of a sequence of *predicates*, separated by commas, which form a logical AND expression. A selection rule succeeds if one of its alternatives succeeds (i.e. leads to the unique identification of some document instance).

A selection rule that – when activated – would cause the configuration of an experimental system (*"select the newest version of all components that I am working on; select the newest published version of all other components"*) might look like:

```
exprule:
        *.c, attr (author, $(LOGNAME)), attr (state, busy);
        *.c, attrge (state, published), attrmax (version).
```

Another example illustrates how known versions of particular modules could be configured into otherwise experimental systems:

```
special_rule:
        afs_def.h, attr (version, 8.22);
        afs_hparse.c, attr (version, 8.17);
        *.c, attr (author, $(LOGNAME)), attr (state, busy);
        *.c, attrge (state, published), attrmax (version).
```

In alternatives, the first predicate (e.g. `*.c`) is usually a pattern against which the name of a document that has to be retrieved is matched. The patterns used by **shape** have the same structure as those used by `ed(1)`. The other predicates allow to express certain requirements with respect to the attributes of documents. *Attr* and *attrge* are predefined predicates that require a specified attribute to be equal or greater-than-or-equal (with respect to a defined or *natural* order) a given value. The similarly predefined predicate *attrmax* requires documents to have a maximal value in the specified attribute. To provide a limited support for handling of *variants*, **shape** also has build-in predicate *attrvar* which affects the setting of some special macros, described later. In order to identify exactly one document instance by evaluation of a selection rule, the alternatives must be sufficient to single out one element from a possible set. Usually, the last predicate of an alternative should guarantee a unique selection. Predicates like *attrmax (revision), attr (state, busy),* or *attr (version, 4.2)* are examples of such selections.

shape checks whether the target already exists and is *current*, or tries to produce it from its *dependents*. A target is considered current if neither the source nor the production context have changed. A principal role for the retrieval of documents is played by their name attributes. In order to check whether a given target is current, both the target document itself, and all of its dependents, must be configured. A name† that has to be configured is passed as implicit parameter to the active selection rule. During the evaluation of the rule, the name is sequentially matched against the name pattern of the rule alternatives. If a pattern matches a name, **shape** attempts to complete the following predicate sequence. If an alternative fails, **shape** will go on and try the next alternative until the rule is completed.

After a name match, **shape** performs an AFS retrieve to initialize an internal *hit set*. The subsequent sequence of predicates will be applied to the set of found AFS objects. An alternative fails if one of the following conditions is true:

- the pattern does not match
- a predicate of the alternative fails
- the cardinality of the hit set is not equal to one, i.e. no unique name resolution was possible.

The application of predicates to the hit set results in the removal of all AFS objects that do not conform to the specified requirements from the hit set. A predicate's evaluation fails if the cardinality of the hit set becomes equal to zero. After the target and all of it's dependents have been configured, the source version ids and the transformation environment are evaluated against the derivation attributes of the derived objects (if present) that would have to be produced.

Selection rules can be activated on a per production basis by simply giving the name of the rule as the first dependent of a production. Thus, it is possible to define targets for the configuration of e.g. test systems and releasable systems within the same **shape**-file.

```
test: exp_rule prog
release: rel_rule prog
prog: x.o y.o z.o
```

A selection rule remains active until it is superseded by activation of another selection rule or until the target that caused the activation is produced. If no selection rule is specified, the default rule, which is the same as Make's ("*select the busy version in the current directory*"), is active.

The following pictures give an overview of the **shape** process.

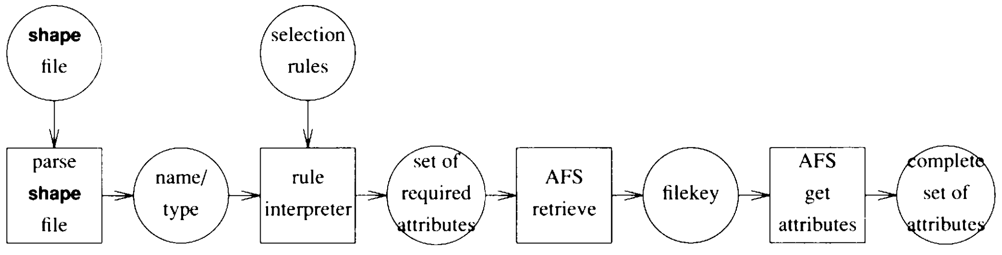


Fig. 3.1: Identification

After having uniquely identified the system's components, the necessary transformation actions can be performed.

† Conceptually, name *and type* of a document are passed as implicit parameters. In this discussion, *name* stands for the concatenation *name.type*

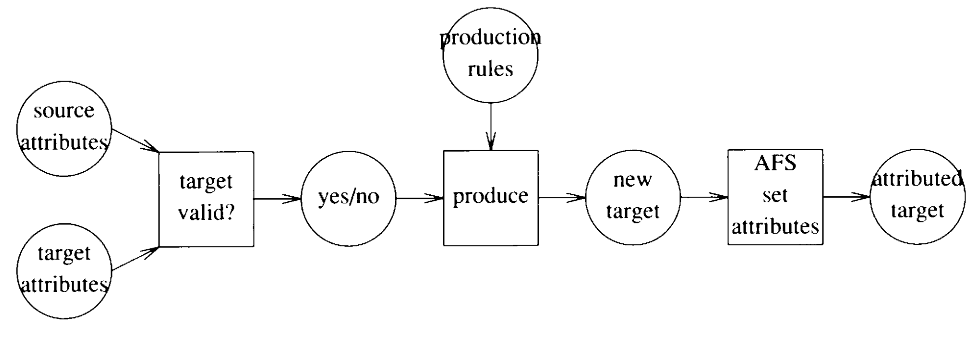


Fig. 3.2: Production

The search space for retrieve operations is either defined by the *current project* (an explicitly selected work context), or by an environment variable that describes the search hierarchy in a fashion that depends on the underlying data storage system. If **shape** cannot find a document in a programmers workspace, it tries to connect to a *project server*, whose address is also defined by the current project. The project server might reside locally or somewhere in the network. It provides controlled access to the project's database. Thus, if a particular document could not be tracked down locally, it might still be somewhere in the project library.

3.4. Transformation Rules

It is part of Make's philosophy, that a transformation produces exactly *one target at a time*. As long as all instances of source and target objects are versionless, there is no big problem with *side effects* (creation of *additional* objects not specified by the transformation rule — $\TeX$ for example, creates a transscript and an auxiliary file along with the dvi-file), because resulting files are automatically updated. Make, just looking at name and modification time wouldn't reproduce these files, if subsequently any dependency on them should be encountered. **shape**, however, basing its decision whether to activate a transformation on a bigger set of attributes, has to *explicitly mark* each produced object with the attributes describing the characteristics of the transformation. In order to avoid overhead transformations, **shape** needs to know the complete set of resulting objects.

To give **shape** a chance to find out about a transformation's significant characteristics, it was necessary to change concept and syntax of transformation rule definitions with respect to Make. In particular, **shape** needs information about the flag– (macro–) definitions that affect the transformation, and names of all objects that are going to be produced. Make's syntax for specifying a default transformation rule doesn't provide for either information. For **shape**, we choose to employ a definition syntax, similar to that used by *cake* [14]. Cake's transformation specifications are based on *name variables*. For example, a possible rule for compiling C programs is

```
%.o: %c
            cc -c %.c
```

with `'%'` as variable symbol. Transformation specifications are actually *templates* for an infinite number of dependencies, each of which is obtained by consistently substituting a string for the variable `'%'`. When an object has to be produced, **shape** determines which transformation rule applies, by matching the object's name against all tokens to the left of the colon. In this matching process, `'%'` is treated as wildcard character. Once a matching token has been found, the `'%'` character is consistently substituted throughout the transformation rule. In the example above, a request to produce an object *matchit.o* would result in a virtual Shapefile entry

```
matchit.o: matchit.c
            cc -c matchit.c
```

The resulting set of source and object names is taken to be the *specification* of the transformation carried out by the shell script associated with the rule. **shape** doesn't know, what is actually done by the shell script, but *assumes* that *all specified objects will be produced* by the transformation, each depending on all source objects and relevant flag– and macro definitions. Another element of transformation rule definitions that hasn't been mentioned yet, is the specification of the macros that are applied in the rule's shell script. To specify them, the rule specification is extended by another, optional colon which separates the flags and macro names from the source names. Thus, a complete transformation rule definition to create a .o object from a yacc source looks like

```
%.o   : %.y    : $(YFLAGS) $(CFLAGS)
                 $(YACC) $(YFLAGS) %.y
                 $(CC) $(CFLAGS) -c y.tab.c
                 rm y.tab.c
                 mv y.tab.o %.o
```

3.5. Configuration Identification and Reproduceability

shape can, for the purpose of *configuration identification*, be asked to produce a configuration identification document (CID). CIDs as produced by **shape** include:

- all components' version identification (composition list)
- the components' variant identification (if present)
- the version of the Shapefile
- the *project domain*, in particular the projectname and the next-higher domain name in which the projectname can be resolved (e.g. a network-, host- or pathname)
- identification of all tools and their versions that were involved in building the configuration (this applies – of course – also to **shape**)
- all macro definitions that were imported from the environment
- the current date, and
- the identification of the programmer who created the configuration.

In order to provide for large configurations that consist of more or less independently maintained subconfigurations, CIDs may also contain references to corresponding *sub*-CIDs. CIDs are the ultimate and most precise description of a configuration. They are designed to describe the circumstances of the configuration identification and build process completely. CIDs are themselves derived objects containing more information than the derivation attributes described above are able to hold. Because this information is crucial for the maintenace of releases, CIDs are subject to version control. The version id of the CID defines the version id of the described product configuration.

Complete reproduceability of system releases is a goal that is hard to achieve. To rebuild old configurations as identically as possible does not only require to keep all document versions that have ever been part of a release, but also a straight history of all tools (e.g. compilers) that have ever been used to produce releases. The cost for this might be considerable administrative overhead, because every project needs a current *resource registry*, where all production tools are logged.

shape has a built-in *rebuild function*. Rebuilding of system configurations is a prerequisite for product maintenance, in particular bug-fixing and customization. For the rebuilding of a release, **shape** may be fed with a CID. It will retrieve the identified **shape** file version and start the system build based on the component versions specified in the CID and the system architecture described in the **shape** file. Whether **shape** will complain about mismatching tool versions, will depend on the grade of accuracy to which a rebuild is done. We believe that this – like a couple of other aspects – should be subject to customizations.

3.6. Controlling variants

There is a common understanding of variants as *alternative* realizations of the same concept. Each alternative has its own revision history, and so there is not necessarily a temporal relationship between them. The typical example for the genesis of variants is porting of software products to different architectures. The idea of variants sounds quite simple, but to actually handle them can be an extremely difficult task.

In C, the concept of variants is most frequently implemented by marking code sections to be conditionally compiled, depending on definitions supplied in a special header file or as command line flags from within the Makefile. For other programming languages, such as Modula-2, variant modules have to be physically separated† and must be stored in different directories. Both techniques are understood by **shape**. To provide means for systematic variant control, **shape** allows to define variant names. These are associated

† you can, of course, also use `cpp` or `m4` ...

with variant-flags that will be passed to the transforming tool, and a variant-path that extends the search space by directories which could host a document variant. The following example illustrates the use of variant definitions:

```
#% VARIANT-SECTION
vclass system ::= (vaxbsd, munix)
vaxbsd:
        vflags="-DUNIX -DBSD42 -DSTDCC -DPSDEBUG"
        vpath="sys/vaxbsd"
munix:
        vflags="-DUNIX -DSYS5 -DPCSCC -DNOVAX -DPSDEBUG"
        vpath="sys/sys5"
debug:
        vflags="-g -DDEBUG"
#% END-VARIANT-SECTION
```

Variant names, as defined in the variant section, can be applied in selection rules by using the predefined selection predicate *attrvar*. They provide a unified concept for the administration of variants within the version control system, with preprocessor technique, and with physical separation. Configuration selection rules making use of **shape**'s variant handling might look like:

```
fsexp:
        afdelta.*, attr (state, saved), attrvar (munix),
                attrmax (revision);
        af*.*, attr (state, busy), attrvar (debug),
                attrvar (munix), attrvar (unixfs);
        make*.*, attr (generation, 4);
        *.*, attrvar (munix), attr (state, busy).
```

shape uses the special macros *vflags* and *vpath* to hold preprocessor options, and extensions to the default document search path. Locations specified by *vpath* are searched for documents prior to the default location. If a document with the specified attributes is found in a *vpath*-directory, the default directory will *not* be searched.

Initially, both macros have an empty value. When variants are selected in an alternative of a selection rule, the associated macro values are *added* to *vflags* and *vpath*. In our example, during evaluation of the second alternative of rule *fsexp* (`af*.*,`), *vflags* would become:

```
vflags="-g -DDEBUG -DUNIX -DSYS5 -DPCSCC -DNOVAX -DPSDEBUG -DFS"
```

and *vpath* would be:

```
vpath="sys/sys5:data/unixfs".
```

Upon completion of each alternative both macros are reset.

The construct *vclass system ::= (vaxbsd, munix)* defines a *variant class*. Variant classes define mutually exclusive variant names. Variant classes can usually be given meaningful names, because they mostly correspond to certain properties, in which the particular variants vary. A variant class *cputype* for instance, with element names *IBM4381, VAX630, VAX7XX, m68k* a.s.o. might be defined in order to prevent the selection of different cpu-specific modules with mismatching cpu attributes. **shape** will complain if such a condition is detected. **shape** encourages systematic description of variant properties in the variant section of the Shapefile. Flag– or macro definitions that are related to variants should be defined in this section rather than places like *CFLAGS*. So far, this is the only concept in **shape** for ensuring semantic consistency across variant system components. In this area, more work needs to be done.

4. Conclusion

At first glance there seems to be no big difference between the functionality of **shape** and a reasonable integration of Make and RCS. However, by integrating version control and system building, we were able to eliminate a number of problems that are extremely difficult to handle with Make and RCS. Each element of the **shape** toolkit utilizes the available data (e.g. "save" operations can be performed on whole systems; production steps can be described more precisely). Furthermore, the implementation of the AFS interface provides a well defined entry point to the version control system that can also be used by other than the toolkit's application programs. In fact, we believe that the AFS facility is general enough to be used in completely different contexts but version control.

The most visible difference between **shape** and Make/RCS is the enriched Makefile – now called Shapefile – with special syntax for the definition of system variants and version selection rules. With the simplicity but expressiveness of its version selection rules, **shape** makes it easy to build and maintain consistent system configurations from a possibly huge number of different source document versions.

Nevertheless, there are still many edges where further work has to be done.

shape does not support dynamic dependencies; a feature that Make users often miss. Because inter-module dependencies can change between versions, the problem of version control and system building had to be tackled first. When adding support for dynamic dependencies, it is important to do this in a *programming language independent* way in order to maintain the openness and generality of the Make concept. We are thinking about a well defined interface to application supplied, language dependent *closure operators*. Some interesting results in this area have been presented with the *Adele* system [5].

Another important feature that **shape** still lacks, is a concept for modularity of Shapefiles. As this implies language concepts that go way beyond what current Shapefiles are able to express, we will discuss the issue in the context of a complete review of the concepts we choose for the first implementation of **shape**.

The variant support which is described more detailed in [11] has also only rudimentary character as it primarily aims at practical life support for the software developer. However, we feel that in this area a lot more work is due.

5. Acknowledgement

This work is part of the **UNIBASE** project. The project is supported by the Bundesministerium für Forschung und Technologie (Federal Ministry for Research and Technology) under grant ITS 8308.

References

1. APOLLO, *Domain Software Engineering Environment (DSEE) Reference,* Apollo Computer Inc., Chelmsford MA., July 1985.
2. AUGMAKE, ''An Augmented Version of Make,'' *System V/68 Support Tools Guide*, pp. 21-40, Motorola Microsystems Inc., Ord.No. M68KUNSTG/D1, January 1983.
3. Erik Baalbergen (erikb@cs.vu.nl), ''Thoughts about 'Make' – Summary,'' *comp.unix.wizards*, USENET, November 1987.
4. Edward H. Bersoff, Vilas D. Henderson, and Stanley G. Siegel, *Software Configuration Management,* Prentice Hall, Englewood Cliffs, N.J., 1980.
5. J. Estublier and N. Belkhatir, ''Experience with a Database of Programs,'' *SIGPLAN Notices*, vol. 22, 1, pp. 84-91, ACM, Palo Alto, California, December 1986.
6. Stuart I. Feldman, ''MAKE - A Program for Maintaining Computer Programs,'' *Software - Practice and Experience*, vol. 9,3, pp. 255-265, March 1979.
7. Glenn S. Fowler, ''A Fourth Generation Make,'' *Proceedings of the USENIX Summer Conference*, pp. 159-174, USENIX asc., Portland, Or., June 1985.
8. Andrew Hume, ''Mk: A Successor to Make,'' *Proceedings of the USENIX Summer Conference*, USENIX asc., Phoenix, Ariz., June 1987.
9. IEEE, *IEEE Standard Glossary for Software Engineering Terminology,* IEEE, New York, N.Y., February 1983.
10. David B. Leblang and Gordon D. McLean, Jr., ''Configuration Management for Large-Scale Software Development Efforts,'' *Workshop on Software Engineering Environments for Programming-in-the-Large*, pp. 122-127, GTE Laboratories, Harwichport, Massachusets, June 1985.
11. Axel Mahler and Andreas Lampen, ''shape – A Software Configuration Management Tool,'' *Proceedings of the International Workshop on Software Version and Configuration Control*, German Chapter of the ACM, Grassau, West-Germany, January 1988.
12. Wolfgang Obst, ''Delta Technique and String-to-String Correction,'' *Lecture Notes in Computer Science*, Springer Verlag, Berlin, September 1987.
13. Marc J. Rochkind, ''The Source Code Control System,'' *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 364-370, 1975.
14. Zoltan Somogyi, ''Cake: A Fifth Generation Version of Make,'' *Australian Unix system User Group Newsletter*, vol. 7, no. 6, pp. 22-31, April 1987.
15. Walter F. Tichy, ''RCS - A System for Version Control,'' *Software - Practice and Experience*, vol. 15,7, pp. 637-654, July 1985.
16. D.M. Tilbrook and P.R.H. Place, ''Tools for the Maintenance and Installation of a Large Software Distribution,'' *Proceedings of the USENIX Technical Conference and Exhibition*, pp. 223-237, USENIX Association, Atlanta, Ga., June 1986.

Appendix A: A sample Makefile

```
#  Transformation rule definitions and
#  configuration management related rules and macros


        @-if [ -s $(SRCDIR)/$*.c ] ; \
        then : ; \
        else \
        echo temporarily checking out $*.c --- $(VID); \
        (cd $(SRCDIR); $(CO) $(COFLAGS) $*.c) > /dev/null; \
        $(CC) -c $(CFLAGS) $*.c; \
        rm $*.c; \
        fi;

        @-if [ -s $(SRCDIR)/$@ ] ; \
        then : ; \
        else \
        echo checking out $@; \
        (cd $(SRCDIR); $(CO) $(COFLAGS) $@) > /dev/null; \
        fi;

SRCDIR = /u/shape/apps
VPATH = RCS
USERID  = `/usr/ucb/whoami`\@`hostname`   # should e.g deliver 'axel@coma'
TMPNAME = .scm-stuff

#  Tool definition & location part
#  (these macros might be site and/or system-dependent)

SHELL = /bin/sh
RCSPATH = /usr/local

CC = cc -DCFFLGS='"$$Flags: $(CFLAGS) $$"'
CI = $(RCSPATH)/ci
CO = $(RCSPATH)/co
RLOG = $(RCSPATH)/newrlog  # added option '-y' - print only revision number
IDENT = $(RCSPATH)/ident
RCS = $(RCSPATH)/rcs

NGFLAGS = -f -l -q -m"New System Generation" -s"Stable"
NRFLAGS  = -q -l -m"This version is part of a release" -s"Release"
NRCIDFLAGS = -l -s"Frozen"

COFLAGS = -q $(VID)
LOGFLAGS = -h

#  Configuration definition part

CFLAGS = -g -DUNIX -DVAX -DUNIXFS -DJOBCONTROL -I$(AFSINC)

#  Installation specific part

INSTALDIR = /u/shape/bin

#  Product definition part

COMPONENTS = save.c dosave.c retrv.c doretrv.c mkattr.c project.c sighand.c \
             util.c save.h retrv.h project.h afsapp.h save.1 retrv.1

PROG = shapetools
PRODUCTS = save retrv
VERSION = version
```

```
MYINC = save.h project.h
SAVEOBJS = save.o dosave.o
RETROBJS = retrv.o doretrv.o mkattr.o
COMMON = project.o sighand.o util.o version.o
ALLOBJS = $(SAVEOBJS) $(RETROBJS) $(COMMON)

AFSLIB = /u/shape/lib/libafs.a

AFSINC = /u/shape/src/inc

#  Component dependencies

all: save retrv

save: $(SAVEOBJS) $(COMMON) $(AFSLIB)
        cc -o $@ $(SAVEOBJS) $(COMMON) $(AFSLIB)

$(SAVEOBJS): save.h project.h afsapp.h

dosave.o: $(AFSINC)/afs.h

retrv: $(RETROBJS) $(COMMON) $(AFSLIB)
        cc -o $@ $(RETROBJS) $(COMMON) $(AFSLIB)

$(RETROBJS): $(AFSINC)/afs.h retrv.h project.h afsapp.h

$(COMMON): $(AFSINC)/afs.h afsapp.h

project.o: project.h

$(ALLOBJS): Makefile

install: all
        install -c -o axel -g unib save $(INSTALDIR)
        install -c -o axel -g unib retrv $(INSTALDIR)
        (cd $(INSTALDIR); rm -f Save; ln save Save)

#  These rules are all configuration management related, and serve to
#  prepare system generations and releases.  It should not be necessary
#  to modify them for different products, as long as certain conventions
#  are obeyed: Define the main-product name in the PROG macro.  List
#  all components' names (i.e.  each revisable entity) in COMPONENTS.
#  Names of individually produced subtargets should be listed in PRODUCTS.
#  The main target should be 'all'.
#
#  This Makefile sample is intended for 'single Makefile systems'.
#  To apply the proposed SCM scheme to more complex systems, some
#  more rules and conventions have to be defined.
#

release: preprel logconf1 all logconf2

logrelease: logconf1 logconf2

generation:     # assumes that all COMPONENTS have a busy version
        @incr $(SRCDIR)/.genno
        @/bin/echo Declaring generation `cat $(SRCDIR)/.genno` for \
        $(PROG) system
        @rm -f .relno
        @-(cd $(SRCDIR) ; \
        $(CI) $(NGFLAGS) -r`cat .genno`.0 $(COMPONENTS))

preprel:
        @incr .relno
        @/bin/echo Preparing release `cat .relno` in generation \
        `cat $(SRCDIR)/.genno` of $(PROG)
        @-(objdir=`pwd` ; cd $(SRCDIR); $(CI) $(NRFLAGS) Makefile \
        $(COMPONENTS) > /dev/null 2> .diag ; \
        awk '{ if ($$2 == "warning:") print $$6; \
        if ($$2 == "error:") print $$5 }' .diag > .ddiag; \
        for i in `cat .ddiag`; \
        do \
        v=`$(RLOG) -y $$i`; \
        echo marking $$i\[$$v\] ; \
        $(RCS) -n"$(PROG)_v`cat .genno`r`cat $$objdir/.relno`":$$v \
        $$i >  /dev/null 2> /dev/null ; \
        done ; \
        for i in $(COMPONENTS); \
        do \
        if grep $$i .ddiag > /dev/null ; \
        then : ; \
        else  \
        v=`$(RLOG) -y $$i`; \
        echo marking $$i\[$$v\] ; \
        $(RCS) -n"$(PROG)_v`cat .genno`r`cat $$objdir/.relno`":$$v \
        $$i >  /dev/null 2> /dev/null ; \
        fi; \
        done ; \
        rm -f .ddiag .diag )

logconf1:
        @/bin/echo Conf-ID: $(PROG) version `cat $(SRCDIR)/.genno` \
        release `cat .relno` of `/bin/date`, > $(PROG).cid
        @/bin/echo 'char *version () { static char ConfID[] = \
        "'`cat $(SRCDIR)/.genno`.`cat .relno` '('`date` by $(USERID)')"; \
        return ConfID; }' > $(VERSION).c
        @$(CI) $(NRFLAGS) -n"$(PROG)_v`cat $(SRCDIR)/.genno`r`cat .relno`" \
        $(VERSION).c > /dev/null 2> /dev/null;
        @/bin/echo by $(USERID) >> $(PROG).cid;
        @(cd $(SRCDIR) ; $(RLOG) $(LOGFLAGS) Makefile $(COMPONENTS)) > \
        $(TMPNAME);

logconf2:
        @/bin/echo Logging configuration in $(PROG).cid;
        @(cd $(SRCDIR); $(IDENT) $(PRODUCTS) | awk '{ if ($$1 == "$$Header:") \
        printf ("     Header: %s %s %s %s %s %s\n",$$2,$$3,$$4,$$5,$$6,$$7); \
        else print $0 }') >> $(PROG).cid;
        @-$(IDENT) Makefile | awk '{ if ($$1 == "$$Header:") \
        printf ("     Header: %s %s %s %s %s %s\n",$$2,$$3,$$4,$$5,$$6,$$7); \
        else if (($$1 != "$$Flags:") && ($$1 != "$$Log:")) \
        print $$0 }' >> $(PROG).cid;
        @cat $(TMPNAME) >> $(PROG).cid;
        @echo
        @echo '***' DESCRIBE PURPOSE OR DESTINATION OF THIS RELEASE \
        '['`cat $(SRCDIR)/.genno`.`cat .relno`']' '***'
        @echo '(terminate with ^D or single '"'.'"')'
        @$(CI) $(NRCIDFLAGS) -r`cat $(SRCDIR)/.genno`.`cat .relno` \
        $(PROG).cid > /dev/null 2> /dev/null;
        @rm $(TMPNAME)
```

Appendix B: A sample Shapefile

```
#
#  Tool definition & location part

SRCDIR = /u/shape/apps
INSTALDIR = /u/shape/bin

SHELL = /bin/sh
CC = cc

#% RULE-SECTION

fsexp:
        af*.c, attrge (state, published), attrmin (state),
                attrvar (unixfs), attrvar (debug);
        *.c, attr (state, busy), attrvar (unixfs), attrvar (debug);

fsrelease:
        *.c, attr (state, frozen), attrvar (unixfs);

dbexp:
        af*.c, attrge (state, published), attrmin (state),
                attrvar (damokles), attrvar (debug);
        *.c, attr (state, busy), attrvar (damokles), attrvar (debug);

#% VARIANT-SECTION

vclass database ::= (damokles, unixfs)

damokles:
        vflags="-O -DDAMO -DUNIX -DVAX -DJOBCONTROL"
        vpath="data/damokles"
unixfs:
        vflags="-O -DUNIXFS -DUNIX -DVAX -DJOBCONTROL"
        vpath="data/unixfs"
debug:
        vflags="-g -DDEBUG"

#% END-VARIANT-SECTION

CFLAGS = -I$(AFSINC)

#  Product definition part

COMPONENTS = save.c dosave.c retrv.c doretrv.c mkattr.c project.c sighand.c \
             util.c save.h retrv.h project.h afsapp.h save.1 retrv.1

PROG = shapetools
PRODUCTS = save retrv
VERSION = version

MYINC = save.h project.h
SAVEOBJS = save.o dosave.o
RETROBJS = retrv.o doretrv.o mkattr.o
COMMON = project.o sighand.o util.o version.o
ALLOBJS = $(SAVEOBJS) $(RETROBJS) $(COMMON)

AFSLIB = /u/shape/lib/libafs.a

AFSINC = /u/shape/src/inc

#  Product dependencies
```

```
all: fsexp save retrv

release: fsrelease save retrv

dball: dbexp save retrv

save: $(SAVEOBJS) $(COMMON) $(AFSLIB)
        cc -o $@ $(SAVEOBJS) $(COMMON) $(AFSLIB)

$(SAVEOBJS): save.h project.h afsapp.h

dosave.o: $(AFSINC)/afs.h

retrv: $(RETROBJS) $(COMMON) $(AFSLIB)
        cc -o $@ $(RETROBJS) $(COMMON) $(AFSLIB)

$(RETROBJS): $(AFSINC)/afs.h retrv.h project.h afsapp.h

$(COMMON): $(AFSINC)/afs.h afsapp.h

project.o: project.h

install: release
        install -c -o axel -g unib save $(INSTALDIR)
        install -c -o axel -g unib retrv $(INSTALDIR)
        (cd $(INSTALDIR); rm -f Save; ln save Save)
```

OFS — An Optical View of a UNIX File System

Paulo Amaral

GIPSI-SM90†
c/o INRIA
BP105
78153 LE CHESNAY CEDEX
FRANCE
mcvax!inria!gipsi!paulo

ABSTRACT

The design and implementation of the Optical File System (OFS) is described. It was conceived to run under UNIX and to deal with Optical Disks. We explain our view on how to develop a file system, at UNIX user level, with a WORM (Write Once—Read Many) device. OFS manipulates multiple file versions automatically. It also works upon an implementation of atomic transactions: fault tolerance implications are studied. Finally, we describe our experience using the OFS by means of a backup utility, that has been used by our software research team since October 1987.

1. Overview

The continued development of Optical Storage Media encouraged our work on this subject during the last few years at GIPSI-SM90. Having worked with the OFS for some time, we describe our view on how to use an Optical Disk and develop a UNIX file system (UFS).

The Optical Disk (OD) is a new and powerful tool. Being very different from a magnetic disk, it offers, among other characteristics, the random access to fixed size blocks (of indelible storage), slow head movement, and fast continuous read/write. Another important characteristic is the reliability it gives, for a relative large period of time (10 years), on stable storage. We wanted to integrate this tool under UNIX, taking advantage of this last obvious strong point, and overcoming the weakest ones. Perhaps the most difficult problem we had to cope with was the impossibility of updating already written blocks.

Ideally, one would like to connect an OD to a system and begin working, just as if it were an ordinary disk. The physical connection is obvious: one only needs a SCSI interface and a driver that knows how to work with it. If we don't need any special commands (like WRITE-AND-VERIFY) the normal SCSI disk driver can be used. We can have immediate access to the OD as a "raw device", but that's all. Unfortunately there is no possibility of having a virtual connection over the OD because traditional file systems are unsuitable to work on indelible storage and they must be modified somehow.

Various solutions have already been found. A first idea is the development of some specific applications that meet with immediate needs (back-up, mass storage, image archival etc.). We could program a set of basic operations to list (ls) or copy (cp) arbitrary data on the OD, but each one would have to know the internal details of OD data organisation. This isn't our approach because it doesn't maintain the usual UNIX transparency and simplicity. If we had ten different devices, we would also have ten different names for the same command. No, what we need is to extend the semantics of the 'mount' command for this new device.

† GIPSI-SM90 is sponsored by the French Ministry of Research and Technology under the contracts 83-B1032 84-E0651 85-B0524

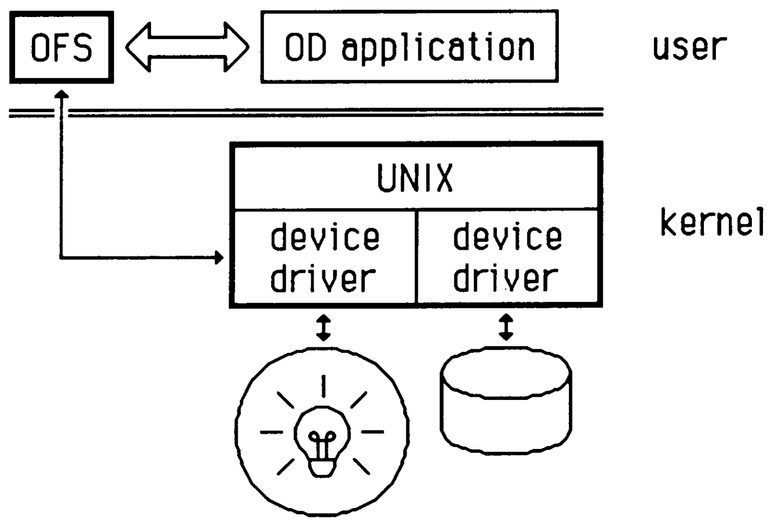


Figure 1

The 'mount' operation always works upon a file system, we will have to use a real one. We could start using a modified UNIX file system, by mapping the data blocks onto an Optical Disk. All inodes, superblock etc. would remain in magnetic media, thus overcoming the major difficulties of an indelible storage. Nevertheless, we measure the reliability of a global system by the reliability of each link. Among other things, if a magnetic disk is always to be used, the system will be less reliable.

Our proposal is to design a file system using an OD only. That's what the OFS actually is. In this way, applications can be programmed neat and transparent. Moreover, existing applications could be immediately used, provided that they work with a standard file system, and that the OFS exists inside the UNIX kernel (or an equivalent and better solution).

We decided to develop the OFS outside the kernel in an user process. This has several advantages including debug facilities. After, we would program some useful applications which work upon it, and which would test OFS. We show the actual implementation of OFS and its interaction with UNIX in figure 1.

To implement the integration the OFS in UNIX, we could use a "file system switch" like the one provided by the VFS (Virtual File System designed by SUN) that has already been ported on our own UNIX (SMX) [ROGADO-86]. Another approach is to work with an NFS compatible interface to the network, and implement the OFS in a complete independent module, at user level. We are presently working on this last solution. A scheme of a future OFS integration in UNIX is shown in figure 2.

We are not going to detail this last item here because it is not completely finished yet. For now, we will concentrate on OFS internals and its applications.

After describing the OFS's general characteristics and data structures which handle indelible storage, we give an overview of atomic transaction implementations in the OFS. Finally, we describe the OFS's recent use by means of a back-up application and a file system debugger.

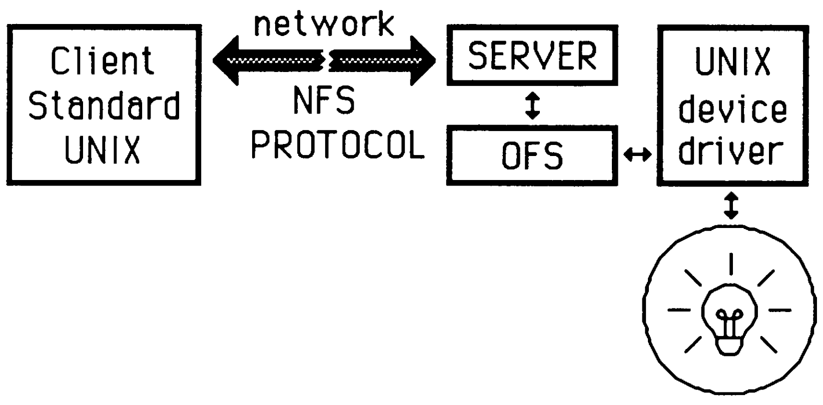


Figure 2

2. OFS General Characteristics

User level compatibility with traditional UNIX file systems is our major concern. The OFS implements a hierarchical directory structure, as usual. Taking advantage of the WORM characteristic of the Optical Disk, we can maintain the successive versions of a file in a natural manner, thus adding a new concept: multiversion files. A file is in fact a directory of historical versions.
Example:

```
/users/gipsi/paulo/dummy:2
/users/gipsi/paulo/dummy
/users/gipsi/paulo/*:*
```

The first path represents the second version of the "dummy" file, the second one, the last version of "dummy", and the last path represents all files and versions of the directory /users/gipsi/paulo. In order to implement this, we need a way to map directories, files, versions, data and superblock (or its equivalent), onto an Optical Disk. All of this without losing the usual flexibility. We thus have to overcome the main difficulty that we mentioned above: the little flexibility of a Write Once storage device.

2.1. Main Goals

Interactive work with the OD is desired. Main applications are somehow related to mass storage (Optical Storage). We are looking towards a file system where:

- files are not deleted frequently
- access to any version must be fast
- adding a version must be inexpensive
- large version handling is a must
- directory access should not be slow
- reliability is fundamental

No space for damaged files. The philosophy here is: Once Written Never Lost (OWNL). If trade-offs are to be taken into account this will be a major one.

2.2. A dynamic Data Structure for the OFS

Most traditional UNIX file systems see directories as simple files storing file names and corresponding inode numbers [BACH], Although this view is changing due to remote file system implementations. Such a directory implementation is unpractical to operate on an OD. The read/write head movements, back and forth, from directory inode to directory data, are too slow. Furthermore, each time we change the directory contents we would have to replicate all directory data. Clustering the directory's inode block with directory contents avoids excessive OD head movements. At the same time once we get a directory's inode we get also its data. OFS doesn't implement traditional inodes but has equivalent structures which we simply call as "nodes".

As far as we know, the solutions pointed out for OD data handling, used several methods for organising information, like Write Once B-trees (WOBT; [MAIER-82] [EASTON-86]), to offer fast random access of data. But with a WOBT data structure space overhead can be substantial due to directory data replication. Besides, directory data can span over several disk blocks in a highly dependent way, which is a major inconvenience for us (as we will see in section 3).

Our solution is a **'ointer fill-in data structure'** for the OD [RATHMANN]. Space on disk is assigned in advance by allocating empty tables. This open structure expects to be filled step by step, and is built upon this simple update basis. Each update action does not need to replicate data: it just fills a precise bucket within a precise table.

For determining where to fill the tables, we use external hash coding on file names [GONNET]. We think it fits the strengths and weaknesses of the OD very well. It gives us a way to know in advance where a certain file is placed, and it compensates us for not being able to make updates afterwards. It requires only one access to retrieve a file node with a well studied hash function, and a carefully dimensioned hash table. Individual files or versions can also be accessed without retrieving all the directory's contents. The only static elements of this solution are table sizes, and we have to be careful with them. It seems logical to use bucket size tables [STAVROS-87]. Although, this is only important if we really want to save disk space or improve listing speed for sparse directories.

Overflows are not chained. We simply reallocate several hash tables for file name storage in directories. A directory is seen by OFS as a set of hash tables (a set as little as possible; there is a trade-off between wasted space and speed). The hash function is a direct conversion between file names and integer values, followed by a modulo division of a prime number. We took a large group of commonly used file names to study a well behaving hash function.

We also use tables for version management. Each table can be seen as a dynamic allocated array. With an infinite array, we can always get an arbitrary version with only one access. As we don't want to lose space, we allocate an array by buckets. Bucket dimension can be arbitrary. Due to OD's characteristics it seems logic to have track size buckets (track dimension depends on the OD). A bucket holds a set of version nodes and a node is hold in a block (the minimal amount of write once data). And so, version nodes are grouped in these buckets, which are linked as binary trees to allow search functions of O(log2): a file with 1000 versions will never require more than bucket 6 accesses!

Finally, we could gather complete file path names, in hash tables also. Although, in this case, directory data clustering seems hard to perform, because we can't predict how many directories will overflow in the same bucket.

2.3. Access Policies

During the development of the OFS we tried to formalise the number of accesses needed to perform a particular task, mainly: file or version retrieval, path traversing and directory listing. They guided our research towards an efficient OD indelible data structure. Each modification to the structure would correspond to a new formula, which would then be analysed. We present the formulas obtained (at the time of this writing) for each operation.

We define a short access as the time of head displacement added to the time of one bucket reading (one track normally). A long access is also the time of a head displacement, but added to the time spent on a continuous read of contiguous buckets (several tracks). This long access is a function of the amount of directory data clustering. This is highly dependent on the machine we are using and cache memory size for data interchange with secondary devices. They depend also on the disk we are using, and they have to be studied accordingly.

Directory data retrieval (DR)

$DR = \int(Fl/Bt)+1$ long accesses

Directory path retrieval (DD)

$DD = \int(Dn/Bd)+1$ short accesses

Path traversing (PT)

$PT = \int(E(DD)*Pl)+E(Kd)$ short accesses

Version retrieval (VR)

$VR = PT+DR+\int(2x log 2((Vn-Kc)/Bv))$ short accesses

E()	=	average
Fl	=	Number of files in directory
Bt	=	Directory table dimension
Dn	=	Number of directory names in directory
Bd	=	Directory bucket dimension
Pl	=	Path length
Kd	=	Constant of accesses for root node retrieval
Vn	=	Version number
Kc	=	Directory constant (always less then Bv)
Bv	=	Version bucket dimension

Average results give one short access for each path component, and $O(log 2)$ accesses for version retrieval (but only one last access for a little number of versions, say 20).

2.4. Write Error Handling

Good or bad, a block can never be updated after it has been written. A ''pointer fill-in structure'' like ours can give some headaches with errors on writing. For this reason we also reserve some space in advance to be used instead of a damaged one.

Each Optical Disk has a known probability of errors on writing. The probability of having a crash in the middle of a block sync is very low: saving one block for errors on each table should be more than enough.

3. Atomic Transactions within OFS

An important thing to be kept in mind is that we are only looking from a file system´s point of view. We are not working with crash proof machines nor stable processors. For us even the operating system can crash. We assume that a system crash can happen at any time during OFS processing.

Having presented things in this pessimistic way, we are going to define the various degrees of atomicity, the mechanism of transactions, and finally, a study on fault tolerance.

3.1. Do we have a Stable Storage?

Yes. Cartridges for the Optical Disk are indelible by individual blocks. We have no need for more than one disk, nor for complicated algorithms to achieve it. We just take advantage of the qualities of this new tool. This leads us to the definition of the low level degree of atomicity.

The minimal amount of data that we can interchange with an OD is a block (1k Byte normally). It has the beautiful particularity of being well written or not written at all. Once well written, it remains stable for a large period of time (10 years). There are no such events like the ''occasional change of the storage medium''. So, it is stable and this particularity of ''all-or-nothing'' makes the writing of one block really atomic.

3.2. Atomic Transactions on Files

OFS is a multiple version file system. Files are always recoverable if the atomicity of each update is assured. There is an unique correspondence between a file operation and an atomic transaction. Each file operation proceeds as follows:

- file locking for writing

- attempt to perform the operation
- validation
- file unlocking

An interrupt on the normal sequence before validation implies a return to the previous state of the file. This is done by simply forgetting the uncompleted operation. A system crash just after validation will have the same effect as file unlocking, causing no harm. These atomic operations (or transactions) remain atomic as long as validation is also considered to be atomic. As validation takes place by writing a single block to disk (and this is atomic as we saw), we achieve also this upper level of atomicity.

This simplicity is only possible by the use of our ''pointer fill-in data structure'', already detailed. If we had to write several blocks to perform a validation, like complete directory data updating for example, things would be much more complicated. This would be the case with other kinds of data structures, where we would have to preserve state in a non trivial manner. It would also be much harder to deal with the ''intentions list'' (see next section) and the commit phase of a transaction [SVOBODOVA-84].

3.3. Global Transactions

We adopted the expression ''global transactions'' as a collection of atomic transactions on files.

A global transaction starts by writing a ''begin transaction mark'' on stable storage. Then, atomic transactions are performed as requested. An intentions list is built upon already completed atomic transactions. Each atomic validation corresponds directly with an entry to the intentions list, and they will be committed at the end of the global transaction. This is tricky, but at the same time a simple method (we dare say that simple is beautiful!). We end up with three different data components in a single atomic block, which are therefore written in a single atomic operation:

- new version descriptor
- directory entry
- intentions list entry

This clustering is done by careful mapping of each virtual table structure:

- directories are maps of several files,
- files are maps of versions and
- the intentions list is also a map of file versions.

As we will see, everything above has a tremendous impact on fault tolerance. We also studied clustering of OFS data structure blocks, as suggested by [STAVROS-87], very carefully. This in order to improve performance, but keeping priorities on directories.

At the end of a global transaction, the commit phase is done by writing an ''end transaction mark'' to disk, and forgetting the intentions list, thus committing finally all atomic entries. Free space on disk is saved at the same time. As this deserves also some reflection, the next section explains it in detail, at the same time that we will explore the measures taken against faults.

4. Free Storage Management and System's Fault Tolerance

It may seem rather strange to associate such different things in the same section. It just reflects our aim to cluster different operations in single atomic ones, thus avoiding the need for preserving intermediate states.

4.1. Tricks

The state preserved on stable storage is of fundamental importance. As we said, we overcome the need of complicated algorithms to keep state coherent, by clustering of operations and data. State is updated when validation of atomic operations takes place, (it is in fact the same operation). A crash at any time will forget any uncompleted operations, and the intentions list will then be used (it memorises all completed atomic transactions).

At the beginning of every global transaction, OFS verifies if it is recovering from a crash (it searches for an end transaction mark), and we get the free space on disk at the same time as well (we can't have a superblock). To do so, we adopted the following solution: there is a special file that contains transaction marks, free space information, and others.

This file is special because, once again, there is a clustering: data is embedded in file descriptors. Data and descriptors are written also in a single atomic write. There is one version of this file for each global transaction. On the other hand, this file is treated by OFS as an ordinary one, being a simple matter of data mapping. Using the data structure that we have already described, many versions of this file should not be a problem. Finally, by writing this file to disk at the end of a global transaction, we can write in a single atomic operation:

- Special file version descriptor
- Special file data containing free space management
- End transactions mark (end of commit phase)
- Special file directory entry

Once again, only a ''pointer fill-in'' data structure can achieve such a degree of clustering, thus obtaining this extreme simplicity. This is at least as important as fast file search.

4.2. Recovery Algorithms

We use state restoration as the recovery method [ANDERSON]. Upon a detection of a recovery state, at the beginning of a global transaction, a file system check is started automatically. Structural errors are eliminated from system's state by:

- recomputing free space,
- getting already validated part of intentions list and
- appending it to the current transaction.

Now we have everything we need and operations can continue normally. OFS forgets uncompleted transactions thus losing already written space on disk. This lost space, added to the time that we lose in the file system check, are the price we have to pay for this reliability.

If an error occurs during the writing of a file structure block (mainly validations) things could be dangerous. Such errors are treated as crash errors. OFS tries to replace badly written blocks by new ones on a special place of the corresponding table. If it doesn't work, it just forgets the operation signaling an error.

What we can say about OFS's fault tolerance is:

- if there is a crash it stops working
- those crashes are tolerated since there are no corrupted files
- time and space is lost to get this reliability (upon a crash, only)

We think that users are prepared to pay this little price for having, for example, a reliable mass storage system. That's why they are working with an Optical Disk anyway.

5. Using OFS

As we already said, OFS complete integration in UNIX is not part of this paper. We offer access to OFS by library routines that make the development of specialised applications possible. Among these, we have a backup utility and a debugger that has been used to test and improve it. Everything runs at user level, including OFS.

5.1. A UNIX command Simulator

Big name for a little thing. This was programmed to give a helpful file system debugger. We called it ''mount_glasses'', to see this Optical thing differently. Its commands are those supplied by UNIX (ls, pwd, cp, cat, etc...). It gives us the funny feeling of working directly under UNIX (and through the SHELL).

There are various levels of entry commands. It has a very low level, similar to fsdb, and the highest one explained above. In between, we have several routine calls to test each file system's layers. It allows us to do a very powerful debugging that has been extremely useful. Any modification to the OFS is intensively tested without much trouble.

5.2. A User-friendly Optical Back-up System

This is a serious application for a change. It has been working regularly at GIPSI-SM90 since October 87. It has given us a way to study the behaviour of OFS under heavy load: the terrible hands of a UNIX software research team.

We picked up cpio´s interface, added file name expansion (as made normally by the SHELL), file version number expansion, and applied the whole on OFS. The result: this extremely simple back-up tool that everybody uses easily. It provides multiple version file access and automatic incremental back-up over standard UNIX paths.

Although far from a real file system, this application satisfies all our basic back-up needs. It is also used by our graphic research team to store images on the Optical Disk.

5.3. OFS future as a Network Server

This is our current work. We want to provide network access to the OD in a completely transparent way. This means that users will be able to ''mount'' an OFS file system over a network. This will be developed as a user-level network server that will provide full compatibility with a standard protocol for remote file systems.

By standard protocol we mean a currently used one, like NFS. It means that machines working with NFS will have access to your server without any modification. Working over file systems on the OD will then be as transparent as NFS.

We avoid the necessity of really embedding OFS in UNIX kernel, which is already too big. Besides, this gives us a complete modularity and much better debug facilities. Compatibility with very different kinds of machines is also a main issue.

We expect to be able to say more on this in a near future.

6. Conclusion

This is mainly an implementation work.

A new file system for working with Optical Disks was presented here. Key features, like dynamic data structures for multiversion file system implementation on the Optical Disk were analysed. Moreover, internal important issues, like clustering of multiple data update operations in single atomic ones, were also presented.

Some applications based on this file system are presently running at GIPSI-SM90. We are going to develop the OFS further, towards a completely transparent UNIX network file system working with Optical Disks.

7. Acknowledgements

I would like to thank José Rogado for a great help during this work, especially when things went wrong, and for his constructive criticism of this paper. Thanks also to Bernard Laborie, José Alves Marques, Robert Jan Honing and Teresa Branco for helpful comments during the writing of this paper. Guy Vaysseix, Patrick Duval and José Legatheaux Martins also gave me some good ideas. Special thanks are due to the GIPSI-SM90 staff for using and testing my work.

8. Bibliography

[ABRA-85]
J. F. Abramatic, ''The SM90 Workstation'', *First IEEE Conference on Computer Workstations Proceedings* , San Jose Nov 1985

[ANDERSON]
T. Anderson and P. A. Lee , *Fault Tolerance Principles and Practice*, Prentice Hall International

[BACH]
Maurice Bach, *The Design of the Unix Operating System*, Prentice-Hall International Editions

[EASTON-86]
Malcolm C. Easton, *Key-Sequence data sets on indelible storage*, IBM J. RES. DEVELP. Vol. 30, No. 3, May 1986

[GONNET]
Gaston Gonnet et al., *External Hashing with Limited Internal Storage*, Proc. of the ACM Symposium on Principles of Data Base Systems, Mar 82

[KNUTH-73a]
D. E. Knuth, *The Art of Computer Programming*, Vol. 1 *Fundamental Algorithms*, Addison Wesley

[KNUTH-73b]
D. E. Knuth, *The Art of Computer Programming*, Vol. 3 *Sorting and Searching*, Addison Wesley

[MAIER-82]
David Maier "Using Write-once Memory for Database Storage", *Proceedings of the ACM Symposium on Principles of Data Base Systems*, March 1982

[RATHMANN]
Peter Rathmann, "Dynamic Data Structures on Optical Disks", *Proceedings of the IEEE Data Engeneering Conference*, Los Angeles 1984

[ROGADO-86]
Jose Rogado and Guy Vaisseix "Yet Another Port of NFS on a System V Based Workstation", *EUUG Conference Proceedings*, Autumn 86

[STANDISH]
Thomas A. STANDISH, *Data Structure Techniques*, Addison Wesley

[STAVROS-87]
Stavros Christodoulakis, "Analysis of Retrieval Performance for Records and Objects Using Optical Disk Technology", *ACM Transactions on Computer Systems*, Vol.12, No.2, June 1987

[SVOBODOVA-84]
Liva Svobodova, "File Servers for Distributed Operating Systems", *Computing Surveys*, Vol. 16, No 4, December 1984

[TANN]
Andrew Tannenbaum, *Operating System Design and Implementation*, Prentice-Hall International Editions

[TANN-85]
Andrew Tannenbaum and Robert Renesse "Distributed Operating Systems", *Computing Surveys*, Vol. 17, No 4, December 1985

Software Re-engineering using C++

Bruce Anderson
Sanjiv Gossain

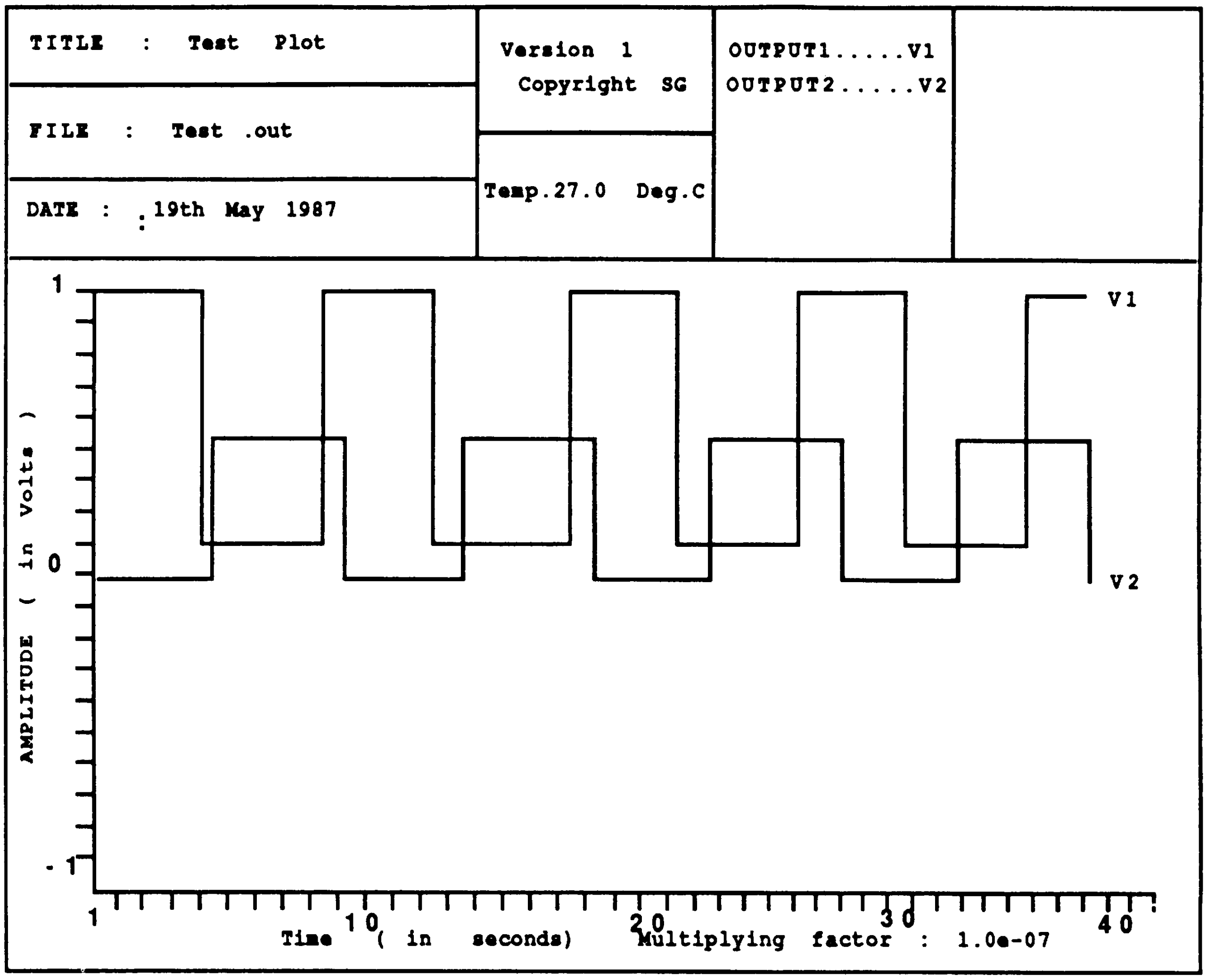
Electronic Systems Engineering
University of Essex
bruce@ese.essex.ac.uk,
goss@ese.essex.ac.uk

The plan for our experiment was to take a piece of software and rewrite it in C++. We wanted the program in question to be locally-generated, written in C, widely used and to be a generic program, one which was typical of a class of programs that were either actually written or likely to be needed. Our idea was to proceed in small steps and to reflect on each step.

Enquiries soon produced a program from our CAD group, one in constant use to take output data from the Berkeley SPICE circuit simulator and output it on either of two HP four-colour plotters. Typical simulation results are tabular data:

```
* TEST FILE *

.PRINT TRAN V(102) V(101)

*SIGNAMES VOUT1 VOUT2

*** TRANSIENT ANALYSIS TEMPERATURE = 27.000 DEG C

TIME            V(102)          V(101)

0.00E-07        0.00e+00        1.00e+00
0.50E-07        0.00e+00        1.00e+00
1.00E-07        0.00e+00        1.00e+00
1.50E-07        0.00e+00        1.00e+00
2.00E-07        0.00e+00        1.00e+00
2.50E-07        0.00e+00        1.00e+00
3.00E-07        0.00e+00        1.00e+00
3.50E-07        0.00e+00        1.00e+00
4.00E-07        0.00e+00        0.10e+00
4.50E-07        0.50e+00        0.10e+00
5.00E-07        0.50e+00        0.10e+00
5.50E-07        0.50e+00        0.10e+00
```

with a corresponding plot:



TITLE : Test Plot

FILE : Test .out

DATE : 19th May 1987

Version 1
Copyright SG

Temp.27.0 Deg.C

OUTPUT1.....V1
OUTPUT2.....V2

The code's generic nature was clear, as members of the group wanted to be able to make different plots and to use other plotters and possibly other simulators.

The idea underlying the reprogramming in C++ was to produce a "kit of parts" from which modified, extended or similar programs could be produced. Thus we were testing out some of the features of object-oriented programming:

- encapsulation (via the object/message mechanism) to control access to representations
- inheritance (via subclassing) to allow modification and extension as construction methods
- automatic storage allocation to encourage safe use of dynamic structures

We took the plotting program and looked at it. It was the first C program written by an experienced but self-taught BASIC programmer, and perhaps that is the key to its construction. The plotter is driven via a serial port (UNIX file) with character sequences. The overall style of the program took the form of "this-then-this", a long sequence of low-level operations; somehow focussed on the production of these character sequences. The mental model associated with it seemed to be very flat, with no hierarchy of concepts.

In particular we noted:

- poor layout
- few meaningful subroutines/functions
- several large functions with no arguments

- many global variables: too many unregistered connections between functions and between sections
- plotter commands widely scattered: over 100 of them, many being repetitions
- lots of static storage: all the storage that could be needed was reserved at the beginning, so that the program would be much smaller if dynamic allocation from stack and heap were used
- poor choice of variable names
- locally inefficient: repeated calculations
- hard to follow: very flat, no structure
- lots of magic numbers: not just a lack of symbolic names (via #define) but no indication of the derivation of sizes
- poor locality: questions to the user about data and formatting were spread around the program; reading anything required jumping around the listing
- size – reduced from 1600 to 1200 lines

Many of these observations are about modularity, so that for example altering the program would be extremely difficult. There is also an issue about documentation, but if the structure were clearer it would be less necessary.

It became clear that we should rewrite the program completely, that although we would use the same output format and thus the same scaling calculations, there was no inherent structure to be extracted; we had to impose our own.

At this stage the programming proper (which we take to include the design) began. We worked from both ends, looking both at the input (the structure of the data) and at the output (the structure of the graphical presentation). One of us (Bruce) was experienced in object-oriented programming and acted as consultant and tutor to the other (Sanjiv) who was responsible for the detailed design, coding and compiling. The most important point in this process was that the class structure provided a focus for the discussions of the design. We talked about classes and protocols. Although there was code in the machine at this stage there was almost no program – we talked about the objects and their relationships and properties, not about processing. The files of class definitions provided the medium of communication between us. Somehow having such files in compilable form was much more powerful than having a specification in English. After some iteration we came up with the following structure.

The classes we decided to use form a rather flat hierarchy:

```
        graphical_object
                box
                division
                        vertical_division
                        horizontal_division
                axis
                        log_axis
                        lin_axis
                graph
        data
        plotter
        plot_request
```

A key decision was how to deal with the concept of space. In our system a box has a visible outline, and contains a single graphical object. In order to divide the space inside a box, the box must contain a division which divides the box into two parts either horizontally or vertically, each containing a graphical object. So we have chosen to describe the layout with a hierarchical binary tree rather than say a set of things at the same level. This is somewhat more complex for the programmer, and rules out some layouts, but is simple and powerful, and relieves us from manipulating constraints about overlapping, adjacency and sizing. As an example, the graphical output shown above is described as:

```
aBox
    in: aHorizontal_Division
        top: aVertical_Division
            lft: aVertical_Division
                lft: aHorizontal_Division
                    top: aText_in_Box
                    btm: aHorizontal_Division
                        top: aText_in_Box
                        btm: aText_in_Box
                rgt: aHorizontal_Division
                    top: aText_in_Box
                    btm: aText_in_Box
            rgt: aVertical_Division
                lft: aText-in-Box
                rgt: aText_in_Box
        btm: aGraph
```

A simple extension to this would allow divisions to contain any number of objects, making the description of the graph flatter – but the software engineering point we want to make is that this kind of discussion is easy to have in an object-oriented language.

What is a plotter? Our plotter object abstracted from many of the details of the HP7225A and HP7475 plotters that were actually used, so that it could draw lines and points and print numbers and strings in a given colour. We retained the plotter's idea about ticks on axes, so that a plotter has big and small tick sizes set, and a member function to draw them.

Who does what? Decisions have to be made about the distribution of responsibility for formatting, scaling and drawing. In our program objects know how to draw themselves on a plotter. They have sizes, and the sizes are set as they are created. For example, putting a graph in a box would involve creating a box of a certain size, and then creating a graph of a corresponding size and putting it in the box. It would of course be possible, though more complex, to allow objects to scale themselves based on the size of the space they were contained in. The drawing itself is straightforward, since every object just draws its own structure, if any, and then requests its contents to draw themselves.

For example, an Axis has an orientation, a legend and a style of marking; it is the subclasses which contain the base and end point, the regular increment and the axis length.

Unlike Smalltalk, everything (for example, data points) isn't an object in our program. This has not seemed to pose any problems, but it might when we try to create programs that are more different than the ones we have considered so far – for example with different kinds of plots.

The coding phase followed. The code was easy to generate in small lumps, using some of the same algorithms as in the original, though many procedures could be simplified or generalised. An example of this was the writing of text within the border of the plot; whereas the original code specified strings and co-ordinates absolutely, the re-engineered code used a general routine with the text string specified separately. The position of the text was calculated according to the box's co-ordinates.

The details of the coding are are not important, but what is important is that the details were easy to take care of because questions arose naturally from the structure and the structure provided a framework for the answers. Thus we discussed ``boxes containing text'' rather than "how to place strings".

The program worked. The only part omitted was the code to interrogate the user as to the data to be used and the plot format, but this was encapsulated into a plot-request object; for our tests we generated a few by hand. This is yet another example of the good structure possible in, and encouraged by, C++.

We attempted to quantify some of the differences between the original and re-engineered software:

- 10 functions with an average of 1.4 arguments and 131 lines of code became 36 functions with an average of 2.9 arguments and 22 lines of code.
- global variables – reduced from 81 to 44
- plotter command distribution – 106 commands spanning 492 source lines reduced to 14 commands spanning 53 source lines
- static storage – 315K reduced to 32K (with dynamic maximum of 282K)

- local efficiency – we removed 12 repeated calculations
- magic numbers – 114 reduced to 25
- locality of user questioning – 529 lines for 21 questions reduced to 60 lines
- size reduced from 1600 to 1200 lines, 5750 to 4000 words
- runtime – about the same

Another good measure of the difference was the cost of modification. It was straightforward for us to alter the layout of the output, for example moving the headings and showing separate rather than overlaid plots for two variables; the thought of doing this in the original terrified us both.

We have presented this work so far as a straightforward investigation of programming and design techniques, in which a careful use of object-oriented programming allowed us to produce a much better piece of software. However, a major output of the project was the learning experienced by one of us who had no previous experience of object-oriented programming or C++.

Sanjiv:

The difference between C and C++ is primarily in the degree of emphasis on types and structure. Stroustrup says: ``The better one knows C, the harder it seems to avoid writing C++ in C style, and thereby lose some of the potential benefits of C++''. As a result of this comment it was with a feeling of excitement and apprehension that I approached this problem.

Having learnt the concepts behind object oriented programming and after experimenting with a few small simple classes and their subclasses, I felt I could approach the matter in hand with some degree of confidence.

A great deal of my time was spent thinking about a possible solution and considering more carefully the various aspects of the program. The approach taken allowed me to break the problem into sections which almost naturally divided the program into classes and their subsequent subclasses. This ``natural modularity'' could only be ascribed to the concepts of object oriented design and C++, which when instilled in the mind changes the view one has of problems and their solutions never previously considered before being exposed to object oriented programming.

I found that I spent a greater part of the time in considering the various aspects of the problem concerned and it's possible solutions than I had done previously when programming in C. This emphasis on the design stage permitted less room for major errors as all the fundamental features of the program were understood completely. I found myself being able to explore more thoroughly any possibilities for a solution by creating test classes and their member functions; I could easily assess their feasibility and could then reject or accept them as candidates for inclusion in my final program. This new approach, I feel gave me a better understanding of the underlying concepts behind the problem and I think it produced a much better all round solution than I could hope to produce had I written my program in C.

An important aspect of the re-engineering design was a critical review with Bruce of each stage upon its completion, and hence any required changes could be made before proceeding on to the next step in our development. These consultations also allowed me to view the problem in hand from a different perspective due to his greater experience and they proved to be invaluable in my learning process. As a result, it was possible for me to prevent any major errors, at an early stage, therefore avoiding complications at a more advanced stage of the program.

The idea of reusability was always present in my mind when giving consideration to any thoughts or ideas I had during my design because of it's importance to modern software engineering methods. Keeping reusability in mind seemed very natural after having learned object oriented design methods and I find I can forsee ways of re-using code for alternative applications much more easily than before due to this new awareness in my thinking. This new awareness can only lead to my writing more practical software solutions than I have done in the past.

Of course learning took place on both sides:

Bruce:

I learned a great deal about C++, and about learning too. I will never forget Sanjiv saying one day ``I never knew programming was like this''.

Our final step was to take the new program to the original author. He had been keen to have us work on his code, wanting very much to improve his C programming. But when faced with our product, his reaction was that he could not conceive of writing programs ``that way'', and that writing them Basic-style was ``much easier''. We would like to undertake a more detailed study of the effort required to achieve a shift towards a more structured approach to software, but this must wait for further work.

However, I do have some speculations on the effort required, based on this project and on experience with other object-oriented languages. Designing good software is difficult, and object-oriented programming puts design up front, so that bad designs are more obvious. Designing good software modules for re-use is even more difficult, and likely to become the domain of special experts. Not everyone can do it, nor can they always afford to hire others, and so makeshift software will continue to be produced and used. Organisations will need to decide where on this spectrum they want to be. Retraining and restructuring to recognise and use this are very difficult.

A UNIX Enviroment for the GOTHIC Kernel

Pascale Le Certen, Beatrice Michel,
Bull Recherche.

Gilles Muller,
IRISA-INRIA
Campus de Beaulieu, 35042 Rennes-CEDEX

ABSTRACT

Creating an operating system on an open machine, implies the use of development methods. This report describes the way chosen to implement the kernel of the GOTHIC distributed system and a well suited environment. Our goal is to design a development system which can run on the successive versions of the kernel. The major advantage of that method is to intensively test the kernel for programming and design errors.

1. Introduction

Development methods are needed to build an operating system on a bare machine. Programs are usually written and compiled on a host operating system while execution and tests are performed on the final target machine. This method is called cross-development.

However we decided to use a different way to set up the kernel of the GOTHIC distributed system. The incremental method we have chosen adds at each new step a new component to the kernel. The V_{i+1} version of the kernel uses the V_i version as its operating system. To achieve this, we provide an independent programming environment which can run on every version of the GOTHIC kernel including the first one, V_0. This method is expected to keep us off using any host system as soon as the V_0 is built.

This paper consists of five sections. Section 2 describes the structure of the GOTHIC kernel. Our method and the development system are presented in section 3. Section 4 details implementation aspects and section 5 gives a brief review.

2. Structure of the GOTHIC kernel

GOTHIC [BANA-86] is a fault-tolerant distributed system. This is based on the notion of fragmented objects and multi-functions provided to express parallel computation. Each primitive of the kernel is atomic (multi-functions are here for) and prevents the propagation of errors in case of problems occurring either at the kernel level or at the user level. Multi-functions solve two main problems for designers of fault-tolerant distributed applications :

- errors are confined in case of bugs in the system,
- global analysis of applications is made easier. Because a multi-function represents a set of cooperating processes, it is easier to understand and "master" the control structure.

To allow the development of applications where speed is critical, the kernel must provide an efficient atomic execution for multi-functions. To achieve this, implemented protocols use a stable storage memory board [BANA-87] which offers atomic primitives (create, read, modify, delete) to access stored objects.

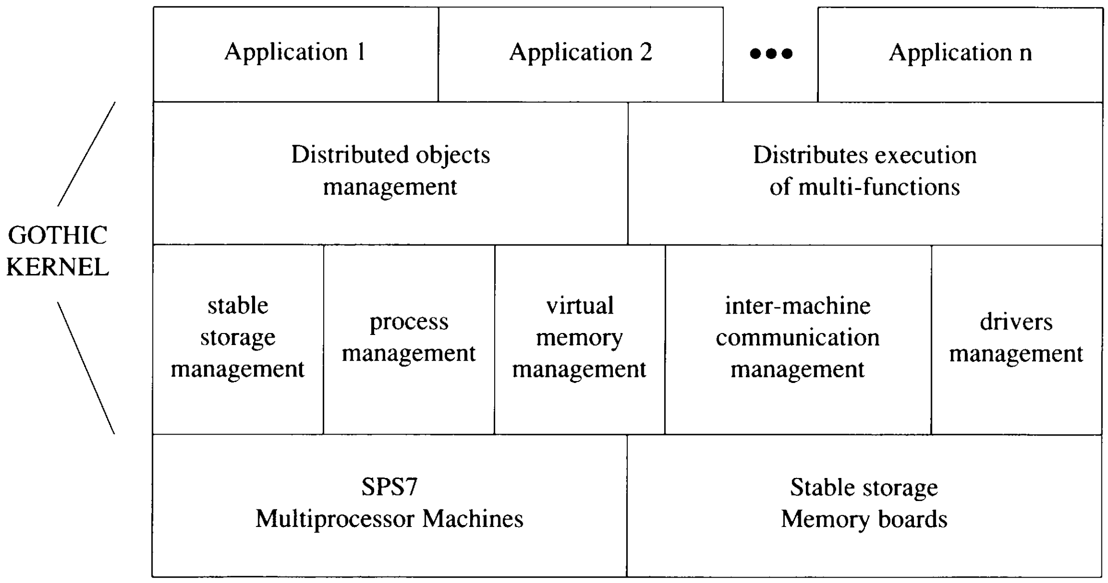

Application 1
Application 2
•••
Application n
Distributed objects management
Distributes execution of multi-functions
GOTHIC KERNEL
stable storage management
process management
virtual memory management
inter-machine communication management
drivers management
SPS7 Multiprocessor Machines
Stable storage Memory boards

Figure 1. structure of the GOTHIC kernel

The structure of the GOTHIC kernel is made out of the following elements:

- several SPS7 multi-processor machines [BULL-85] compose the bare machine, on which we add a stable storage to each processor.
- on this bare machine the first software layer includes hardware managers (stable storage manager [COQU-87], communication manager [LUCA-87], drivers managers, ...) and those dealing with virtual memory and process management. This layer is built on a real-time system named SPART [BULL].
- the second layer is distributed on the whole set of machines. It executes multi-functions and messages objects. We are presently specifying and developing this layer.

3. Development strategy

The GOTHIC kernel is progressively built. If we are forced to use a host system and cross-development methods to produce the initial version (V0) of the GOTHIC kernel, it is no longer the case afterwards. In fact, we find it interesting to build each V_{i+1} version of the GOTHIC kernel using the last V_i version; this means that any potential design or programming error will be rapidly detected. The resulting operating system is thus a true working/usable system and not just a ''model''.

Of course, this is only possible if a proper programming environment is available on this kernel. Programmers should have the possibilities to log-in (!), to write, compile and test programs. This environment can be seen as an application running on the current V_i version of the GOTHIC kernel.

Each V_i version of the GOTHIC kernel plus the environment constitute a stand-alone operating system which can run on the bare machine. This independence keeps us from using a host system for development or tests. The GOTHIC kernel is not slowed down by any lower software layer, and no constraint appears on the design nor the development.

The initial V0 version of the GOTHIC kernel comprises only the SPART operating system and the stable storage and communication managers. As a result, the programming environment is designed to fit SPART and its file management system. However the environment is never a definite one : it has to be adapted to the new elements brought by each new version of the kernel (multi-functions, fragmented objects, ...). When the whole GOTHIC kernel is built, this programming environment will be modified for the last time to become a GOTHIC application.

4. Implementation

4.1. Choosing the programming environment

There are two possible ways of producing a programming environment. The first is to create an integral environment which fits GOTHIC concepts (multi-functions, fault-tolerance, ...) and uses some state-of-the-art facilities in man-computer interfaces (multi-windows, mice, ...). The second way is to adapt the program interfaces or libraries of an existing system, and then to compile the tools and commands using these libraries.

To implement such a large project as the GOTHIC kernel, at least a source code control system and a software release management tool are highly desirable. A minimal but sufficient programming environment will offer the following features :

- user log-in to an access-controlled working space,
- a general file and disk management commands set,
- a full-screen editor,
- a C-compiler,
- a source code control system and a software release management utility.

Let us again notice that this programming environment will have to be completely re-designed to fit the concepts of GOTHIC. A this will require too much time than we can afford *now* for just the production of the environment, we choose the second way ie. the adaptation of the program interfaces of an existing environment. Therefore we only have to re-write the program interfaces of the chosen environment, and then new tools can be added with some modifications.

Many reasons led to the choice of a UNIX System 5.2 compatible environment, including :

- UNIX offers all of the above facilities,
- SPART has a UNIX-compatible file management system (FMS).

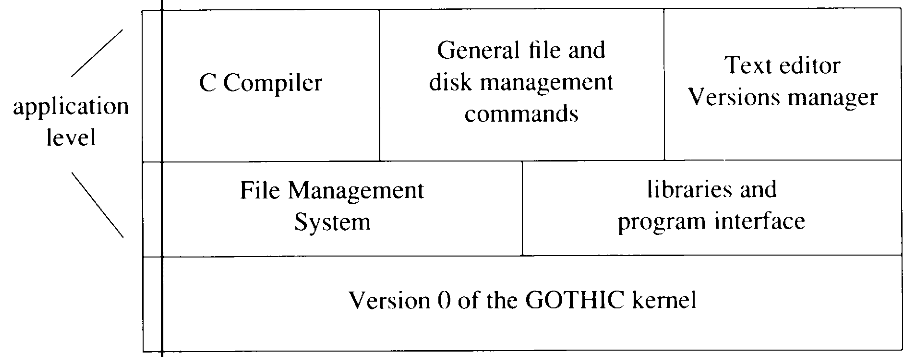

application level
C Compiler
General file and disk management commands
Text editor Versions manager
File Management System
libraries and program interface
Version 0 of the GOTHIC kernel

Figure 2. structure of the programming environment

4.2. SPART system

The SPART operating system offers supports to real-time applications on the SPS7 multi-processor machine. SPART has been designed according to SCEPTRE principles [BROW-84]. SPART can manage up to eight processing units. Each processing unit possesses its own code of the kernel. A task (or a SPART process) is created on a processing unit and stays there until the end of its execution. All requests concerning a task (creation, activation, destruction, ...) are local. With the global memory and the communications kernel it is possible for tasks running on different processing units to share memory, to communicate or to synchronize (semaphores). The heart of SPART is its kernel which schedules tasks, manages signals, mutual exclusion, interruptions, ... and carries inter-processing unis communications. Above the kernel are several agencies which offer higher level devices. These devices constitute the available interface for applications, as basic functions of the kernel are not directly accessible. The ''Object'' is the basic entity int the SPART kernel (there are task objects, program objects, segment objects, ...). Each agency manages a special class of objects; it offers proper primitives for this class of objects. Objects can be stored in either local or global memory. When in local memory they can only be accesses locally, whereas objects stored in global memory are accessible by any processing unit.

The SPART system has an "operator task" allowing users to create and control tasks from the system console by calls to special primitives.

4.3. Development methods

The programming environment must match the version of the kernel on which it runs. The first version of the kernel consists mainly of the SPART system. Therefore low-level libraries have to offer a proper interface to SPART objects; and so they will have to be re-written using the multi-functions concept as soon as these ones are available.

We have to cope with cross-development methods to produce the first version of the environment. Programs are edited and compiled on the host system : SPIX (UNIX System 5.2 of the Bull Company). The machine used is a bi-processor SPS7, with SPIX running on one of the processing units, while SPART runs on the other. These two systems work independently. It is forbidden to share a logical volume in write mode because the cache-memory of each file system lies in local memory. That is why communications between the two systems is achieved by using the same volume but in mutual exclusion. Executable files produced while SPIX is running are stored on this volume; the volume is un-mounted from SPIX file system and then mounted on SPART file system. This way, each file system is kept in a coherent state. The volume is mounted on SPIX file system in a directory called `/gothic`. All object files produced for the environment are copied into this directory in a UNIX-like tree-hierarchy. On SPART, this volume is mounted as the root of the SPART file system; it is directly accessible by all of the commands of the environment. With the operator-task it is possible to mount or dismount the shared volume and to run the environment on SPART.

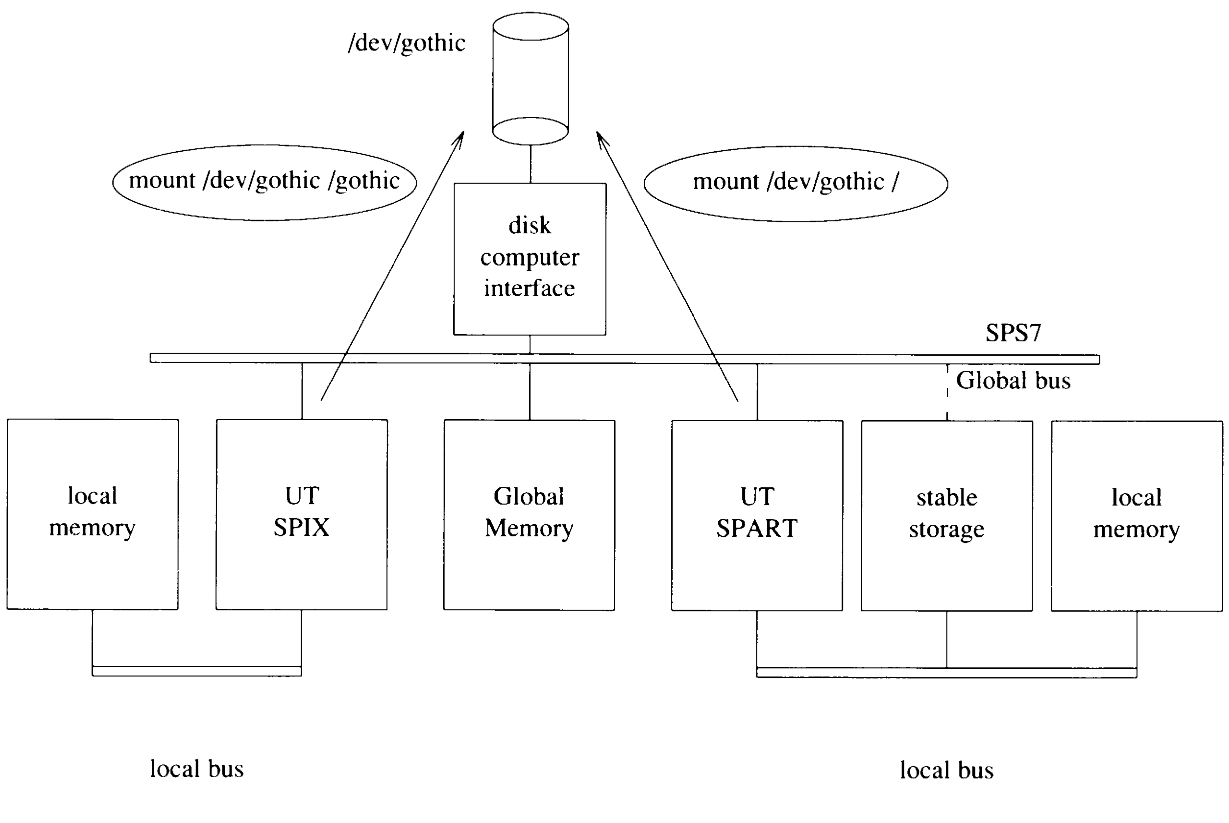


Figure 3. development method

4.4. Implementation of the environment

Out adapted programming environment is System 5.2 compatible. This environment is made out of two parts (see figure 4).

This first part interfaces the kernel and manages UNIX objects such as processes and files. It includes the modified file management system of SPART (FMS), programs libraries (libc, ...) and UNIX primitives which create processes : FORK and EXEC. Several tasks have to be carried out here. First, the file management system (FMS) has to be adapted to fit out needs. Second, as SPART processes are different from UNIX processes, we have to write UNIX primitives to create UNIX-like processes on SPART. Third, we must bring in our environment the libraries used to line the commands.

In the second part, commands and tools available for the user can be found. These two parts are required to implement the environment. We built them either by adapting existing programs or by re-writing machine-dependent primitives.

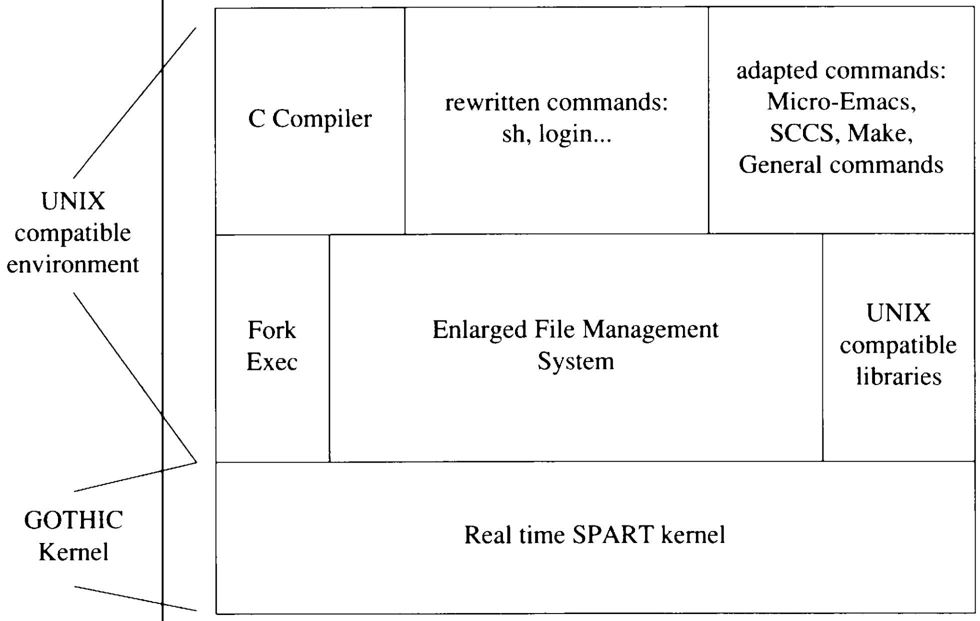

C Compiler
rewritten commands: sh, login...
adapted commands: Micro-Emacs, SCCS, Make, General commands
Fork Exec
Enlarged File Management System
UNIX compatible libraries
Real time SPART kernel
UNIX compatible environment
GOTHIC Kernel

Figure 4. UNIX-compatible environment

4.4.1. Adapting FMS

The FMS file management system is a subset of the UNIX System 5.2 one. Some functions such as "pipes" are missing. Moreover FMS has been designed to enable SPART to use or share files with a UNIX system. This explains why we found some insufficiences in FMS, especially in the management of serial input/output lines. Let us look at an example : some of the missing functions are those which control the way the interface deals with special characters (such as erase, carriage-return), the echo, the data-flow (XON-XOFF), the transmission speed, and so on. In System 5.2, characteristics of a line are stored in a structure named "termio". This structure can be read or set by an "ioctl" command, respectively associated with the "TCGETA" and "TCSETA" function. (see figure 5). In FMS, each control command is accessible through a special "ioctl" function. In order to look like the UNIX interface, "TCGETA" and "TCSETA" functions have been added to FMS standard commands as well as other features, including the typed-in character test, and the transmission of a line-feed with each carriage-return.

UNIX	SPART
ioctl(file-desc, function, arguments)	ioctl(file-desc, function, arguments)
int file-desc;	int file-desc;
int function;	int function;
struct termio *arguments;	struct termio *arguments;
functions	
TCGETA: Reads the characteristics of the serial line	FDX: The line is put in full-duplex mode.
	HDX: The line is put in half-duplex mode.
TCSETA: Sets the characteristics of the serial line.	ECHO: Echoes the received characters.
	NO_ECHO: Suppress the echo mode.
	XON: The traffic is controlled by XON-XOFF
	XOFF: Suppresses the XON mode.
struct termio {	Format of the arguments is function dependent.
unsigned short c_iflag /* input mode */	
unsigned short c_oflag /* output mode */	
unsigned short c_cflag /* control mode */	
unsigned short c_lflag /* local mode */	
char c_line; /* line discipline */	
unsigned char c_cc[8]; /* control char */	
};	

Figure 5. Comparison between ioctl commands

4.4.2. UNIX and SPART process management

4.4.2.1. Context of a process

The context of a UNIX process holds a set of informations about the process and its execution environment. The following informations are available:

- uid, gid : real user and group identifiers. These are inherited from the "father process" (the process which created this one).
- euid, egid : effective user and group identifiers. These identifiers are used to restrict the access rights of the process to files and are also inherited from the father process. These effective identifiers are the same as the real ones (uig, gid) unless access rights have been modified by the process or by one of its ancestors via the EXEC primitive.
- current directory.
- the list of currently opened files.
- the umask variable (added to the file creation rights).
- call parameters : argc, argv of "main", pointing to the command line.
- a UNIX environment, described by a set of variables, each being associated with a character string representing its value. Variables such as terminal type (TERM variable), or search rules (PATH variable), can be defined in this set. The shell associated to any user enables him to read, modify, or enlarge the set of variables. The process accesses this UNIX environment via a character string array pointed to by the "environ" variable.

To implement the UNIX context of a process we use three data structures. The FMS context contains the uid, gid, euid and egid identifiers, the current directory, descriptors of the files opened by the process, and the umask variable. Two other structures are used for the call parameters (ie. the command line) and the UNIX environment. These structures are stored in a segment associated to the process. When a process creates another process or a "son process", this segment is duplicated to let the son inherit the context of its father.

4.4.2.2. Process creation

In UNIX, a new process is created by a call to the FORK primitive (see figure 6). FORK duplicates the process in a way similar to cellular division : the two resulting processes share the code and have identical stack and data segments and context. FORK returns zero to the created process, and the system-identification of this created process to its creator process.

It is the ''create-task'' primitive of SPART which creates SPART tasks. This primitive works on an executable object. This executable object must have been previously loaded in main memory. The created task then waits until its creator task activates it by a call to the ''start-task'' SPART primitive. Afterwards, the created task and its father task no longer are related. This makes the difference in semantic between process creation in UNIX and SPART.

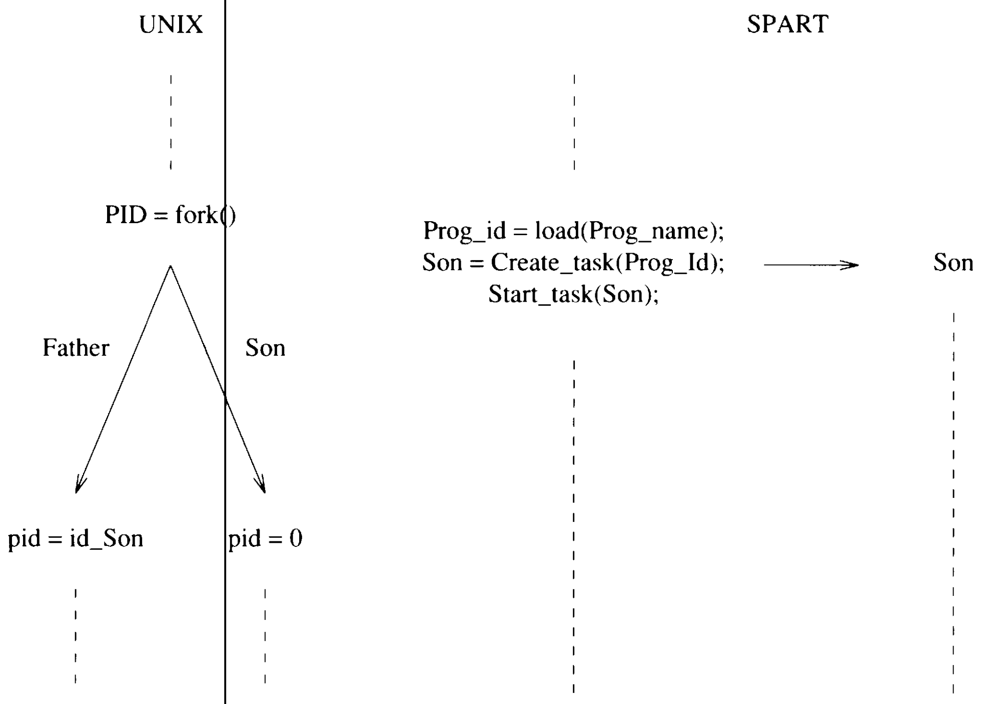


Figure 6. creating a new process with UNIX and with SPART

Of course, the FORK primitive is not enough to build an operating system : FORK can only create a twin process of the calling process (same code). This is why UNIX designers have introduced the EXEC primitive. EXEC creates a new process which has code, stack and data segments all different from those of the calling process. Only the context is inherited from the ''father'' process. EXEC can be called through several names (execl, execv, execve, ...); each of these names corresponds to a different set of parameters.

FMS initializes the real and effective identifiers (uid, gid, euid and egid) of each new SPART task according to the values fixed while generating FMS. All the SPART tasks have therefore the same uid and god. But it is highly desirable, specially in a multi-user programming environment, that the values of these variables correspond to those of the user. To achieve this, FMS fixed values are those of the system administrator; at the very beginning of any task (start up phase), it is possible to update the value with the appropriate one.

4.4.2.3. Implementing process creation

There are two methods for implementing FORK and EXEC. They can be introduced either on FMS, using the usual user interface, or at the FMS level, that is, as a special agency of the SPART kernel. The first method is easier : we only need user primitives. However it is necessary to initialize identifiers with those of the system administrator, which does not ensure any protection to the users. The process could take anybody access rights. The second method requires to work at the kernel level with low-level objects, but then protection is ensured. It is more difficult to implement this latter method because any error could crash SPART. The recovery procedure consists in rebooting the two systems (SPIX and SPART) and in making another version of the SPART kernel.

Actually, both methods have been successfully implemented. In order to be rapidly able to create UNIX-compatible tasks on SPART, we first implemented the first method. A mini-shell was written; when a user types in a command, it creates and activates the corresponding SPART task. This FORK-EXEC chain simulation sonn enabled us to test the UNIX-commands that we had adapted to SPART. Then the other method (see figure 7) was developed, ensuring working space protection.

Implementing FORK

Father process	Son process
A call to fork: – The program counter of the concerned code file is incremented. – A new task object is created. – A memory block is allocated for the SPART context of this new task. – Data, stack and environment segments are created. – The SPART context of the new task is initialized. – The new task can then run, and is scheduled.	
	– Copy the stack, data and environment segments of the father process into those of the son process. – The code segment is mapped in the logic space of the task. – Copy of the file descriptors. – Identifiers (uid, gid) are copied. – Current directory is set. – Umask variable is copied. – Context switch. Return the zero value.
Return the identification of the son process.	(The son process then really runs by itself

Figure 7. implementing FORK

When a UNIX-like process has created a ''son-process'', it can wait for this son to terminate and to give its ''father'' a report about its execution. For example, this is the case of the C-compilation-scheduler ''cc''. Cc calls many programs successively. Any program in this chan can only run if the previous one has returned a correct report value. The UNIX primitive to wait for a process to terminate is ''wait''; a process terminates its execution by a call : ''exit(code)''. We added a ''CR'' variable in the environment segment of processes for the transmission of this ''CR'' report value. When a son process ends by ''exit(code)''. its ''cr'' variable takes the ''code'' value. ''Wait'' is implemented as follows. The father process has to set an access on its son environment, and then wait for its son to terminate by a call to the SPART primitive ''wait_end_task''. Then, the father process can read in its son environment segment the cr report value and deaccess this segment. In the SPART system, a segment can be destroyed only if there is no more access set on it. In our implementation, each task asks for its environment segment to be destroyed when it ends. However, if the father process of this task has called the wait primitive, the destruction can only occur after the father (has read cr and then) has deaccessed the environment segment (because ''wait'' sets an access on the environment segment of the son process). Otherwise, the environment segment of a task is destroyed at the end of this task.

Implementing EXEC

EXEC just keeps the context of the calling process and re-initializes the code, data and stack segments according to the argument used to call EXEC. The sizes of the new program segments can of course be different from those of the calling program. Therefore, it is not sure that the only inherited segment (environment or context) is mapped at the same address in the logical space of the task (see figure 8).

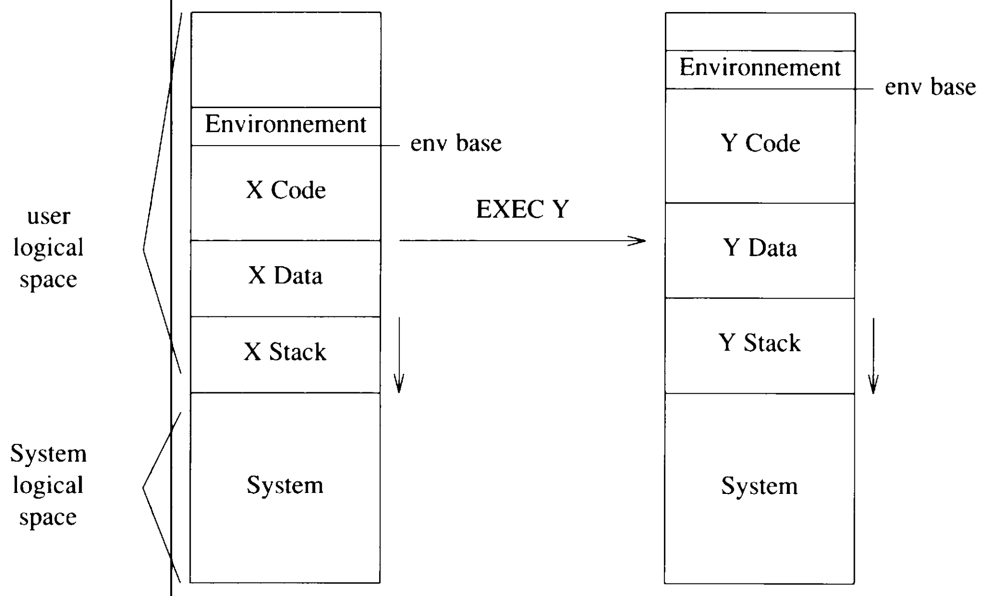


Figure 8. mapping of segments before and after EXEC

The environment segment contains a fixed number of pointers to internal variables (to strings containing the values of the environment variables and to arguments in the command line). These pointers have to be re-computed to take into account the new position of the environment segment in the logical space of the task. Two elements compose a pointer : the base (mapping address of the segment) and the offset within the segment. The conversion equation is:

new_pointer = (old_pointer - old_base) + new_base,
with(old_pointer - old_base) = offset.

Here is a simplified implementation of EXEC :

A call to EXEC implies the following operations:

- Access and execution rights on the new program file are checked.
- Enviroment, code, data and stack segments are un-mapped.
- Stack, data and code segments are destroyed.
- The program counter on the old program file is decremented.
- The new program is loaded.
- The program counter on the new program file is incremented.
- The new program is unloaded. This wiil be done only when no task uses it anymore.
- If an ''S'' bit is set instead of an ''x'' for this file, the effective identifiers are updates accordingly.
- New stack, data and code segments are mapped.
- The kept environment segment is mapped.
- Pointers in the environment segment are converted.
- Return.

Using EXEC, it is possible to change any effective user identifier : euid, egid of a process. If access rights on the executable file have an ''S'' bit set instead of an ''x'' for user or group rights, the calling process will run this code with the effective owner and group identifiers (euid, egid) of the file. Real identifiers (uid, gid) remain unchanged : they are those of the calling process. So, it is possible to have commands which need a larger visibility of the system, such as ''su'' (change to system administrator mode), or which access to a reserved space, such as the spooler ''lp''.

4.4.3. Adapting libraries

Most of the ''libc'' can be adapted to SPART only by a re-compilation of primitives. However, machine-dependent functions or process-management functions have to be re-written.

Signals are not implemented, but ''empty primitives'' were written to produce error-free links. Signals primitives are generally used to prevent the murder of a task when this could disrupt the system.

In UNIX, the function ''time'' returns the number of seconds elapsed since the 1st January 1970. The FMS function ''time'' is not the same, but was used to implement a UNIX-lik etime function on our SPART system.

When a process just starts running, it is not the "main" procedure which is executed first, but instead, a "startup" procedure assigned by the SPART system. Of course, the startup procedure tends to call "main", but before, it initializes the heap, standard input/output files and the UNIX environment of the process. The startup procedure was modified to fit our UNIX-like process management in our programming environment.

Thus we have simultaneously UNIX processes and "pure" processes in our SPART system. There are therefore two versions of the startup procedure, each one integrated in a library (libunix or libnounix). During the link, we use one or the other library, according to the edited task.

Enviroment variables can be accessed through two primitives : "getenv" and "putenv". Getenv gets the value of an environment variable and putenv modifies it. These useful primitives are available in our environment. Finally, the following functions were adapted to our system : libc, libcurses : used to manage terminals, libld : used to produce objects in the right format (COFF).

4.4.4. Running the environment on SPART

There is a task, named "init", that the operator must create and start from the system-terminal to start the execution of our programming environment. This task waits for a user to log-in from a user-terminal. If the user is recognized, a command process or a shell is attributed to him. In the current state of our system, two "init" tasks can be run in parallel and so two users can log-in and work simultaneously.

Shell is a process which recognizes commands typed in by the user and executes the corresponding programs. Because our environment is a subset of UNIX, we decided to write our own "mini-shell", which uses only the UNIX facilities implemented in our environment. Another reason for this choice is that we wanted to test the behaviour of our UNIX-adapted commands whereas FORK and EXEC were not yet integrated into the SPART kernel. Then, the simulation for FORK and EXEC was performed in our shell.

4.4.5. User interface

Some basic commands (df, du, ls, pr, cat, mkdir, rm, ...) are adapted from UNIX V7. Other available commands are UNIX System 5.2 modified commands.

A special "hybrid" case is the C compilation chain. Both V7 and System 5.2 were used to adapt this chain. The chain is made up of a C-compilation-scheduler "cc" which successively calls : a pre-processor "cpp", a compiler "c0", "c2"which optimizes the code produced by c0, an assembler "as", and finally a link editor "ld". Our System 5.2 was designed for a DEC/VAX machine. To avoid re-writing a compiler and an assembler for a 68020 processor, we chose to get the first elements, from "cpp" to "as", by adapting V7 sources (because V7 sources are designed for a 68000 processor). Objects produced by these programs (from "cpp" to "as") are then modified and put in the correct format to fit System 5.2. A "convert" command was therefore inserted in the original compilation chain. Finally, the link editor "ld" comes from System 5.2. Other commands which manage program objects : load, dump, size, s3, nm, ... come from System 5.2 too.

Make and SCCS are used to manage respectively program objects and program sources. The available editor is a reduced version of Emacs : Micro-Emacs 3.7. This editor is practical, not too big, and easy to adapt.

5. Conclusion

Today we have a UNIX programming environment running on the initial version of the GOTHIC kernel. This kernel consists of an enlarged SPART system with stable storage and communication facility (it is the V0 version).

Next versions of the kernel will directly come from this initial version, using the adapted programming environment. Therefore the final version of our system will have been fully tested.

As soon as multi-functions and GOTHIC objects management system are implemented, the actual programming environment and the file management system will have to be re-designed according to GOTHIC concepts. The UNIX-like programming environment will then be a GOTHIC application.

6. References

[BANA-86]

Banatre J.P., Banatre M., Ployette, Fl. *An Overview of the GOTHIC Distributed Operating System.*

[BULL-85]
Bull. *Structure Generale de la SPS7.* Technical report, BULL, January 1986.

[BULL-86a]
Bull. *SPART, Système d'exploitation temps reel Version 21.* Reference manual, BULL, May 1986.

[BULL-86b]
Bull. *SPIX, Présentation Generale.* Reference manual, BULL, April 1986.

[COQU-87]
Intégration d'une memoire stable dans une architecture multi-processeurs. Rapport de fin d'etudes D.E.S.S I.S.A., I.F.S.I.C. Universite de Rennes 1, June 1986.

[LUCA-87]
Lucas C. *Developpement d'un système de communication pour reseau de machines SPS7.* Rapport de fin d'etudes I.N.S.A., June 1986.

[BANA-87]
Banatre J.P., Banatre M., Muller G. *Ensuring Data Security and Integrity with a fast stable storage.* Proc. of 4th on Data Engineering, Los Angeles February 1988.

[BROW-84]
Browaeys F., Derriennic H., Desclaud P., Fallour H., Faulle C., Febvre J., Hanne J.E., Kronental M., Simon J.J., Vojnovic D. *SCEPTRE: proposition de noyau normalise pour les executifs temps reel.* TSI Volume 3 No.1, January 1984, pp 45-62.

Directly Mapped Files

Andreas Meyer

Stollmann GmbH,
Max Brauer Allee 81
D-2000 Hamburg 50
West Germany

ABSTRACT

Directly Mapped Files is a file access method implemented under UNIX System V Release 3.
The entire file appears to the user process like a large byte array in the virtual address space and may be accessed without any read, write or seek operations. Thus, many programs, especially those working on data bases, intermediate files, or complex data structures, may be written much more easily and run faster. The paper describes the implementation in the UNIX kernel and its direct relationship to the demand paging algorithms. An example (*ar*) demonstrates the issues for user programs.

1. Introduction

When I first learned to use UNIX, I read:

> In the UNIX system, a file is a (one dimensional) array of bytes.

a statement from the *UNIX Implementation* by K. Thompson. So why can't we access a file ''as is'': like a long array of bytes?

With *directly mapped files*, a file is mapped into a segment in the virtual address space of a program. Thus the contents of the file are directly accessible. No read, write or seek is required to access the data in the file. This simplifies user programs and eliminates all buffer management problems.

This solution is in no way new: already the MULTICS system only allowed accessing files by mapping them into the virtual storage. Other operating systems had similar implementations.

2. The Open, Close, Read, Write, Seek Interface

Example 1

Copy an array of 100 integers from one record to another.

What we want to do (and this is what can be done with dmf files):

```
for (i = 0; i < 100; i++)
    record2[i] = record1[i];
```

What we currently do, if we believe in C and UNIX:

```
int a;
for (i = 0; i < 100; i++) {
    lseek(fd, record1 + i * sizeof(a), 0);
    read(fd, &a, sizeof(a));
    lseek(fd, record1 + i * sizeof(a), 0);
    write(fd, &a, sizeof(a));
}
```

What programmers will do since they feel the overhead:

```
int a[100];
lseek(fd, record1, 0);
read(fd, a, sizeof(a));
lseek(fd, record1, 0);
write(fd, a, sizeof(a));
```

Thus the programmer makes some optimizations to bypass the kernel overhead.

Of course, this is a very simple examples, and there is nothing to say about the latter solution. But what if the data structures are more complex, e.g. fileds of unknown size, that can only be interpreted by analyzing the data itself, e.g. strings? In such cases, good programmers will try to transfer data by chunks.

Programmers organize data in memory due to what they feel is the best overhead compromise between the clean solution – leaving things to the UNIX kernel – and the reduction of overhead.

The kernel does everything for you to optimize your file access, like reading blocks in advance, or keeping blocks in a buffer cache. If a programmer bypasses these algorithms, or adds his own optimizations, the kernel optimizations can be lost.

When programmers create their own buffering, cache scheme, or whatever, they are cheating the kernel! If they do not, their programs become intolerably slow.

The idea of UNIX is that a programmer must not even think about optimization or file access.

The existing UNIX interface to files works well with stream oriented data. This encourages programmers to organize their data in a sequential manner – even on databases or intermediate files, which will never be sent to a pipe or to a sequential device like a tape.

A disk is a direct access storage medium, but UNIX programmers still write programs that work well on paper tape readers and punches.

3. The DMF User Interface

After opening a file by

```
fd = open(path, flags, mode);
```

another system call

```
buf = dmfopen(fd, [addr], [size]);
```

attaches a new segment to the program.

addr specifies the address where the new segment is to be located in the virtual address space of the user. If not specified the next free address is used.

size is the maximum size the file is allowed to grow to. This is needed to claim space in the user's virtual address space. If not specified, *ulimit* is used instead.

buf returns the start address of the data segment – that is where the file begins in memory. If addr was specified, *buf* will be equal to *addr*.

From now on, the program may access the file by any normal memory operations, as explained in Figure 1.

Since the kernel cannot determine the exact length of the data accessed by the file, the size must be specified in a close instruction:

```
dmfclose(fd, size);
```

The user is responsible for keeping track of the size. If the size is not specified or if no dmfclose is issued, the size is rounded upward to the nearest multiple of the page size.

If several programs open the same file, their segments all map the the same pages frames in memory. If one program writes to the file, the data is immediately available to all other programs.

Of the file was opened *read-only*, a write access causes a *segmentation* fault

```
#include <fcntl.h>

char *dmfmap();

#define Usage { printf("use: %s <file>0, argv[0], argv[1]); exit(2); }
#define Perror(x) { perror(x); exit(2); }

main (argc, argv)
int argc;
char *argv[];
{
        register fd, fd2, size, i;
        register char *ba, *bp;

        if (argc != 2) Usage
        if ((fd = open(argv[1], O_RDWR)) < 0) Perror(argv[1]);
        size = lseek(fd, 0, 2);
        if ((ba = dmfmap(fd, 0, 0)) < 0) Perror("dmfmap");

        for (bp = ba + size, i = size; i; --i)
                *bp++ = *ba++;                  /* copy data */

        if (dmfclose(fd, size+size)) Perror("dmfclose");
        exit(0);
}
```

Figure 1.

The program *dup* uses dmf to duplicate the contents of a file; i.e. the file has the old contents twice after execution of *dup*.

4. The Transparent Solution

The transparent solution allows existing programs to run using dmf with only minor changes: only the open on files working with dmf has to be changed from *open* to *opendmf*. All other operations on the file, like read, write, seek, close, look as usual. The program will be linked together with a module which replaces the normal I/O operations. If these operations pertain to dmf files, they are executed by memory copies, like the read in Figure 2.

Of course, this does not give the full power of dmf to these programs: they still work with small buffers and do not access the file directly in the segment. On the other hand, there is no need to rewrite existing programs, and you can still let them take advantage of dmf. As an example, I took the archiver (*ar*), a well known UNIX utility which handles a library of object files. *Ar* needs replace an object file of 20k in a library of 100k.

It turned out that *ar* running with transparent dmf moved lots of its cpu time from system to user mode. The system call interface was eliminated. Overall the CPU time was reduced and real time dropped by 50%.

Ar runs about twice as fast as before. Better results canbe expected if the entire program is rewritten to handle the library and the object module as arrays in its virtual address space.

	real	user	sys
ar	35.1	0.5	2.8
ar with transparent dmf	15.3	1.5	1.0

```
write(fd, adr, siz)
{
        register struct dmfx *dx = &dmfx[fd];
        if (fd < 0 || fd >= NOFILES) || dx->dmfy == 0)
                return _write(fd, adr, siz);
        memcpy(dx->dmfy->buf + dx->pnt, adr, siz);
        dx->pnt += siz;
        if (dx->pnt > dx->dmfy->siz)
                dx->dmfy->siz = dx->pnt;
        return siz;
}

read(fd, adr, siz)
{
        register int i;
        register struct dmfx *dx = &dmfx[fd];
        if (fd < 0 || fd >= NOFILES || dx->dmfy == 0)
                return _read(fd, adr, siz);
        i = dx->dmfy->siz - dx->pnt;
        if (i <= 0)
                return 0;
        if (i > siz)
                i = siz;
        memcpy(adr, dx->dmfy->buf + dx->pnt, i);
        dx->pnt += i;
        return i;
}
```

Figure 2.

The read and write functions of the transparent solution move data from user buffer to file (and vice versa) using memory copy operations. These functions also support the pointer into the file (dx->pnt) and the size of the file (dx->dmfy->siz).

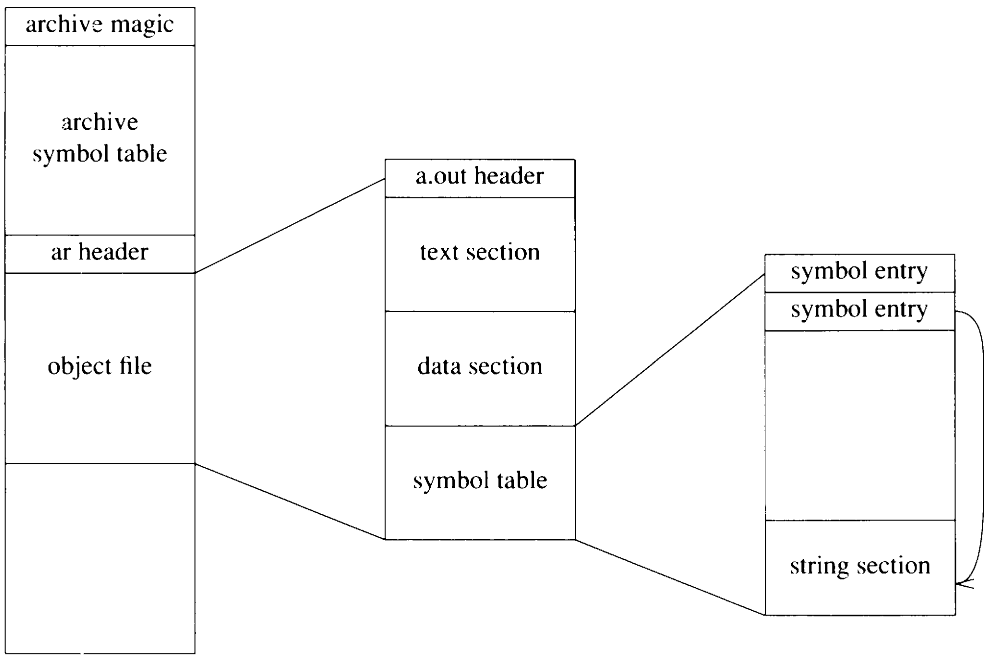


Figure 3.

A complex data structure like an archive of object files is a typical example of prudent use of dmf. *ar r lib obj* needs to access the file more then 2000 times to replace an object in the archive.

(lib=100k, 20 objects, obj=20k.)

5. Kernel Implementation

The virtual address space of a user program is split into segments. Like text, data and stack, files appear like long arrays within the address space.

In a demand paging UNIX – like System V Release 3 – text segments are no longer loaded into main memory before a program starts. When the program tries to access a page that is not available in memory, this page is *faulted in* from the *a.out* file in the file system. Thus nearly all algorithms needed are already available in the kernel. Kernel functions that had to be changed are:

- When the dmf region is built up, its pages map into the file in the file system.
- If a page fault occurs, the data is read from the file into memory. If the page lies beyond the size of the file, the page is cleared to zero.
- If the region is freed, all dirty pages are written back into the file.
- The page stealer vhand rewrites dmf pages to the files.

Only 30 source lines had to be inserted or changed in the kernel code. For the implementation of the new system calls and the read/write of dmf pages, 3 pages of source code were added.

If more than one process accesses the same file by dmf, they share the same segment in memory. As with text segments, if the sticky bit of the file is set, the region will stay in memory, but its pages may be swapped out by the page stealer.

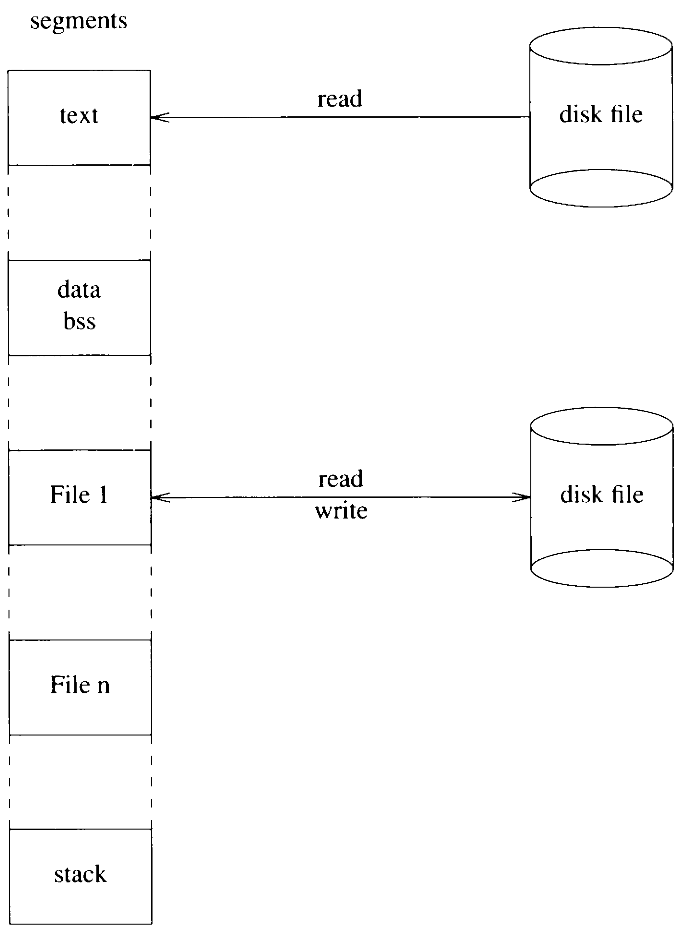


Figure 4.

The virtual address space of a user program is divided into segments. Like text, data and stack, files appear like long arrays within the address space.

6. Suggested Use and Limitations

The size of the file is always a multiple of the page size of the implementation – in our port 2k. Users have to handle the size of files themselves. They can set the file size in the file system when closing the file.

An access beyond the size of the file results in a segmentation fault, a signal which normally aborts the program. Even if a program catches this signal, there is no simple way to handle the end-of-file situation and to continue the program.

Directly mapped files cannot be used with standard input, standard output or pipes. All other data organized as a stream can be handled well by the standard UNIX I/O interface.

7. Conclusions

Directly mapped files is a powerful access method implemented under a demand paging UNIX system such as System V Release 3. It simplifies the file access of user programs as files appear as what they are said to be: just long arrays of bytes. Dmf cannot replace all UNIX files, but it is useful with complex data structures, data bases, and intermediate files.

Dmf essentially reduces the kernel CPU time. With the transparent solution – which requires only minor changes in existing programs – most of this reduction is lost to the user CPU time. In any case, the kernel system call overhead is discarded, context switches are eliminated, and real time reduced drastically.

By moving CPU time from the kernel to the user process, it also makes the implementation of UNIX in a multiprocessor environment easier!

Grep Wars

Andrew Hume

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
research!andrew

ABSTRACT

Subsequent to the Sixth Edition of the UNIX system there have been different versions of the searching tool *grep* using different algorithms tuned for different types of search patterns. Friendly competition between the tools has yielded a succession of performance enhancements.

We describe the latest round of improvements, based on the *fio* fast I/O library and incorporating the Boyer-Moore algorithm. Although *grep* is now 3–4 times faster than it was, *egrep* is now typically 8–10 (for some common patterns 30–40) times faster than the new *grep*.

1. Introduction

In the beginning, Ken Thompson wrote the searching tool *grep*. It selected and printed lines from its file arguments that matched a given regular expression. In 1975, just after the release of the Sixth Edition of the UNIX system, Al Aho decided to put theory into practice, and implemented full regular expressions (including alternation and grouping which were missing from *grep*) and wrote *egrep* over a weekend. *Fgrep*, specialised for the case of multiple (alternate) literal strings, was written in the same weekend. *Egrep* was about twice as fast as *grep* for simple character searches but was slower for complex search patterns (due to the high cost of building the state machine that recognised the patterns).

Ever since, each of the tools has sporadically improved its performance, mostly as a friendly rivalry between the owners of *grep* (Thompson, and later on, McMahon) and *egrep* (Aho). We inadvertently joined this rivalry in mid 1986 by improving *grep*'s I/O management, thus enabling *grep* to leapfrog in front of *egrep* for common (simple) patterns. During August 1987, we improved the performance of *grep* by another factor of 2 and that of *egrep* by a factor of 3. By implementing the Boyer-Moore algorithm for literal strings within *egrep*, we further improved its performance (for patterns containing literal strings) by another factor of 8.

This report describes how these improvements were made. We first describe *fio*, a fast buffered I/O library and compare its performance against the standard I/O library *stdio*. We then describe the changes made to *grep* and *egrep*.

2. The *fio* library

The *fio* library is a small, efficient buffered interface to the UNIX file system. For most purposes, it supplants the older, slower and larger *stdio* library. A full description of the *fio* routines is given in the appendix. A short summary follows in three sections: buffering, input, and output routines. As in the UNIX system calls, all I/O is referenced by file descriptor.

Buffers are initialised by calling *Finit*. The user need not explicitly call *Finit*. Buffers can be written out by calling *Fflush*. Input and output streams can be linked by *Ftie* which causes an output stream to be flushed whenever the associated input stream is read. *Fseek* is analogous to the *lseek* system call taking buffering into account.

Fread reads a specified block of characters and *Fgetc* reads a single character. *Frdline* returns a pointer to the next (newline terminated) line. It does not copy the line into a user buffer like the *stdio* routine *fgets*. *Fundo* undoes the effect of the last *Fgetc* or *Frdline*.

Fwrite writes blocks of characters and *Fputc* writes a single character. Formatted output is done by a new and slightly incompatible implementation of *printf* called *Fprint*.

2.1. Speed comparison with *stdio*

The raison d'etre for *fio* is speed. This is accomplished by eliminating unnecessary copying and by using the *memory* routines for copying (*memcpy*) and searching characters (*memchr*). The implementation is straightforward with few surprises. The library totals 263 lines of C source. (The equivalent part of *stdio* is 545 lines.) The rest of this section compares *fio* and *stdio* routines.

gets/Frdline

This simply tests speed of reading line at a time with no processing. The input had 19883 lines totalling 1024933 bytes.

```
        char buf[4096];
        register char *s = buf;

#ifdef  STDIO
        while(gets(buf) != NULL)
#else
        while(s = Frdline(0))
#endif  STDIO
                ;
```

library	user	sys	total
stdio	19.5	3.2	22.7
fio	3.7	2.6	6.3
stdio/fio	5.27	1.23	3.60

puts/Fwrite

There is no *fio* analog to *puts* from *stdio*. *Fwrite* must count the number of characters to be output. We give times for longer lines (with length `nrand(4096)`) as well.

```
        char buf[4096];
        register n;
        long nb;

        for(n = 0; n < 4096; n++)
                buf[n] = 'X';
        for(nb = 0; nb < 1000000; nb += n+1){
                buf[n = nrand(128)] = 0;
#ifdef  STDIO
                puts(buf);
#else
                Fwrite(1, buf, strlen(buf)); Fputc(1, '\n');
#endif  STDIO
                buf[n] = 'X';
        }
```

library	smaller			longer		
	user	sys	total	user	sys	total
stdio	17.8	4.7	22.5	15.1	4.2	19.3
fio	7.5	3.7	11.2	2.3	3.8	6.1
stdio/fio	2.37	1.27	2.01	6.57	1.11	3.16

fwrite/Fwrite

If the line lengths are known then it might be tempting to use the *stdio* routine *fwrite*. Again, we give times for longer lines (with length `nrand(4096)`)

```
        char buf[4096];
        register char *s = buf;
        register n;
        long nb;

        for(n = 0; n < 4096; n++)
                buf[n] = 'X';
        for(nb = 0; nb < 1000000; nb += n+1){
                n = nrand(128);
                buf[n] = '\n';
#ifdef  STDIO
                fwrite(buf, 1, n+1, stdout);
#else
                Fwrite(1, buf, n+1);
#endif  STDIO
                buf[n] = 'X';
        }
```

library	smaller			larger		
	user	sys	total	user	sys	total
stdio	25.8	4.6	30.4	22.9	4.6	27.5
fio	4.3	3.9	8.2	0.9	3.6	4.5
stdio/fio	6.00	1.18	3.71	25.44	1.28	6.11

Fwrite is so wretchedly slow that using *puts* improves things:

library	smaller			larger		
	user	sys	total	user	sys	total
stdio	17.5	4.6	22.1	15.1	4.2	19.3
fio	4.3	3.9	8.2	0.9	3.6	4.5
stdio/fio	4.07	1.18	2.70	16.78	1.17	4.29

gets,puts/Frdline,Fwrite

This example reads and writes lines.

```
        char buf[4096];
        register char *s = buf;
        register n;

#ifdef  STDIO
        while(gets(buf) != NULL)
                puts(buf);
#else
        while(s = Frdline(0)){
                n = FIOLINELEN(0);
                s[n] = '\n';
                Fwrite(1, s, n+1);
        }
#endif  STDIO
```

Note that the length of the last line read comes for free with *fio*. The input has 5088 lines totalling 313267 bytes.

library	user	sys	total
stdio	10.9	2.6	13.5
fio	2.0	2.1	4.1
stdio/fio	5.45	1.24	3.29

getc,putc/Fgetc,Fputc

This represents character at a time processing. *Fio* loses badly because the *stdio* char I/O routines are inline macros. The input has 5088 lines totalling 313267 bytes.

```
        register char c;

#ifdef  STDIC
        while((c = getchar()) != EOF)
                putchar(c);
#else
        while((c = Fgetc(0)) >= 0)
                Fputc(1, c);
#endif  STDIC
```

library	user	sys	total
stdio	8.8	2.3	11.1
fio	34.8	3.5	38.3
stdio/fio	0.25	0.66	0.29

2.2. Summary

If we compare libraries on the basis on user CPU time, *fio* is about 6 times faster than *stdio*. (The single exception, single character I/O, has not been a problem so far but is not too difficult to fix.) For some programs the speed of I/O is not important. However, for a large number of programs (including many of the commonly used UNIX tools), I/O speed is important if not dominant. The case of *grep* described below is typical; changing to *fio* increased the speed by a factor of 3–4.

3. Tuning *grep*

The primary motivation for tuning *grep* was its slow performance on large (> 10MB) files. (In retrospect, the improvements described below are apparent even for small files.) The limited time available for tuning precluded work on the pattern matching code so we concentrated on improving the I/O management. In particular, we wanted to see what improvement we could get by replacing *stdio* routines by *fio* routines in a typical UNIX tool.

The timing examples are searching for the beginning of a line in a file of 13,931 filenames totaling 512,000 bytes. Because our main concern was I/O performance, we chose a pattern that was cheap to match, thus the execution times reflect as little of the matching code as possible. The timing examples differ only in the amount of output generated. The datafile is 512,000 bytes of filenames of mean length 37 bytes. In each example, every line in the input is matched. The small output example is

```
    grep -c '^' datafile
```

which produces 1 line of output (the number of lines). The large output example is

```
    grep '^' datafile
```

which effectively copies every input line to the output.

The structure of *grep* is fairly simple. The main program loops over its file arguments processing each one in turn. The processing of each file is also fairly simple; the file is opened, each line is read, the pattern is matched against the line and the appropriate action taken on that line. Most often, the action is to print the line if the pattern matched the line. Thus, the original code (*grep.1*) looks like

```
while(gets(buf) != NULL)
        if(match(pattern, buf))
                printf("%s\n", buf);
```

By adopting *fio* input we derive *grep.2* shown below. We simply replace the call to *gets* by a call to *Frdline*. Note that the line is stored inside *fio* 's buffer, not our buffer.

```
while(buf = Frdline(fd))
        if(match(pattern, buf))
                printf("%s\n", buf);
```

The final version (*grep.3*) uses *fio* for output as well. The call to *printf* is converted to a simple *Fwrite* call (note that the newline character is placed where the terminating null was).

```
while(buf = Frdline(fd))
        if(match(pattern, buf)){
                register x = FIOLINELEN(fd);
                buf[x] = '\n';
                Fwrite(1, buf, x+1);
        }
```

The times for each of these versions reflect the superiority of *fio*. Times are given in seconds of user time. As the gains due to improving the output routines will depend on how much output is generated, the times are given for "small" and "large" amounts of output. The speedup is given as a ratio of the time for *grep.1*.

version	small		large	
	time	speedup time	speedup	
grep.1	12.48	1.0	25.54	1.0
grep.2	4.16	3.0	16.47	1.6
grep.3	4.05	3.1	6.7	3.8

The payoff for using *fio* is high; *grep* runs 3.1–3.8 times faster than the original version considering the small amount of code changed.

4. Tuning *egrep*

The original implementation of *egrep* was a deterministic state machine. Performance was often poor because of the potentially exponential time needed to construct the state machine. In 1983, Aho introduced lazy evaluation of the (cached) state transition tables by techniques described in [1]. In practice, as few state transitions are needed, the lazy algorithm ran nearly as fast as the original algorithm (one additional test inside the inner loop) with zero initialisation time.

As we will be tuning the pattern matching code for *egrep* as well as the I/O code, there is also a timing example denoted "matching" that executes the pattern matching code for every byte of the input: `egrep abcd input` (note that no line contains `abcd`).

4.1. Tuning I/O

Egrep.2 replaces the *stdio* output routines (*printf*) in *egrep.1* by the appropriate routines from *fio* (*Fwrite*).

Next, the input code was changed to use *fio* routines. The input code was an elaborate double buffer scheme which complicated the state machine code considerably. The change to *fio* was intended mainly to simplify the code and to measure how much difference there was between the *fio* library and hand-coded I/O. The input was done by the following code:

```
if(--ccount <= 0){
        if(p <= &buf[BUFSIZ]){
                if((ccount = read(f, p, BUFSIZ)) <= 0) goto done;
        } else if(p == &buf[2*BUFSIZ]){
                p = buf;
                if((ccount = read(f, p, BUFSIZ)) <= 0) goto done;
        } else {
                if((ccount = read(f, p, &buf[2*BUFSIZ]-p)) <= 0) goto done;
        }
}
```

The *fio* version (*egrep.3*) is

```
if(*p++ == '\n'){
        if((p = Frdline(f)) == 0) goto done;
        len = FIOLINELEN(f);
        p[len++] = '\n';        /* Frdline nulls the \n, put it back */
}
```

The next version, *egrep.4*, had a careful recoding of the inner loop of the pattern matching code. The execution times are

version	small		large		matching	
	time	speedup time	speedup time	speedup		
egrep.1	4.75	1.0	13.82	1.0	15.95	1.0
egrep.2	4.59	1.0	7.13	1.9	15.03	1.1
egrep.3	2.93	1.6	5.31	2.6	16.55	0.96
egrep.4	3.06	1.6	5.28	2.6	9.99	1.6

Given we have to examine every byte in the input, *egrep* cannot be significantly improved.

4.2. Tuning the Algorithm

Obviously, the real question is whether we can avoid looking at every byte. The answer (in practice) is yes if we are looking for literal strings. A literal string is a regular expression with no meta-characters, that is, every character stands for itself. The Boyer-Moore algorithm[2] performs very fast searches for a literal string. It tries to match the pattern against the input from right-to-left. Failures can often advance the input pointer by the length of the search pattern. For example, if we have a 10 character string not containing a `y` (say), and the current input character is a `y`; then we can advance over 10 input characters as the `y` cannot match any part of the pattern. In general, we can advance the input pointer by the distance of the character from the end of the pattern. If the input character matches, we then do a slow character by character check. The first version of *egrep* implementing Boyer-Moore detects patterns which are literal strings and executes special purpose code (about 164 lines of C, a 27% increase in the number of lines of code) achieving spectacular results, a factor of 8.1 faster.* An unexpected consequence of the efficiency of Boyer-Moore is that asking *egrep* to give the line number for lines that match slows *egrep* down by a factor of 2 because it now has to look at every input byte to count the newlines.

What about regular expressions (rather than literal strings)? We cannot apply Boyer-Moore directly to the regular expression matcher. However, we can use Boyer-Moore to filter out lines that can't match by extracting the longest (that is, the most effective for Boyer-Moore) literal string in the regular expression and running the regular expression matcher only on the lines that match†. The implementation for *egrep.5* involves running a stripped down version of the state machine (29 lines, a 3% increase) over the line that matched the literal string pattern. The times show the improvement over *egrep.4*; as expected, the longer the literal string, the more effective the Boyer-Moore filtering is.

pattern	egrep.4	egrep.5	speedup
`n.*`	3.5s	3.6s	.97
`na.*`	9.7s	3.1s	3.1
`nam.*`	10.0s	1.8s	5.6
`name.*`	10.1s	1.7s	5.9
`name\..*`	10.0s	1.2s	8.3
`name\.c.*`	9.9s	1.0s	9.9

This improvement is not machine specific; measurements on a Sun III workstation and a VAX 11/750 show similar speedups. In fact, searching for a 30 character string (in our standard data file) takes no measurable user time on the SUN. Indeed, for the typical use where the pattern includes a literal string longer than 3 or 4 characters that matches relatively few lines, runtime is determined by how fast the underlying operating system can do I/O.

5. Conclusion

It seems clear that most programs doing substantial I/O can significantly improve their performance by substituting *fio* routines for *stdio* routines (or even hand-coded I/O). For example, *grep* improved its user (cpu) times by a factor of 3.1 to 3.8. These improvements are straightforward to apply and the payback is large (provided I/O is significant). Furthermore, *fio* is highly portable. It provides an invariant user interface (for example, that doesn't depend on the size of integers) and the implementation is simple and easy to port.

In some circles, these changes to *grep* and *egrep* would be dismissed as "simply engineering." It is true that the biggest gains come from better algorithms such as Boyer-Moore, both for literal string patterns and as a filter for regular expressions containing literal strings. (The speedup depends on the pattern but factors of 8–10 are common.) Nevertheless, solving a problem like searching for lines in a file intrinsically involves problems like managing I/O. Indeed, once the algorithms are sufficiently good, these problems dominate.

What is the current state of pattern searching? *Egrep* is about as fast as it can be for matching regular expressions (in practice, a runtime of $O(|input|)$), and it uses the Boyer-Moore algorithm (in practice, a runtime of $O(|input|/|pattern|)$) for literal strings. On the most common patterns (literal strings), *egrep*

*What is not spectacular is how long this took to come about. The Boyer-Moore algorithm was published in 1977. A kludgy but workable implementation of Boyer-Moore within *egrep* was done by Woods at NASA, Ames in 1986.

†Woods' *egrep* actually implements this directly by emitting the matching lines into a pipe feeding the normal *egrep*!

is about a factor of 10 faster than *grep*. On regular expressions like `d...name`, *egrep* is typically 30–40 times faster. The only reason (apart from habit or loyalty) for using *grep* is the feature of back-referencing in the search pattern (referring to the input matched by an earlier part of the search pattern).

References

[1] Aho, A. V., Sethi, R., Ullman, J. D., *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1986, pp121-128.

[2] Boyer, R. S., Moore, J. S., "A Fast String Searching Algorithm," *Communications of the ACM*, **20**, *October, 1977.*

NAME

Finit, Frdline, Fgetc, Fread, Fseek, Fundo, Fputc, Fprint, Fwrite, Fflush, Ftie – fast buffered I/O

SYNOPSIS

```
#include <fio.h>

void Finit(fd, buf)

char *buf;

int Fprint(fildes, format [, arg ...])
int fildes;

char *format;

char *Frdline(fd)

int FIOLINELEN(fd)

long FIOSEEK(fd)

int Fgetc(fd)

void Fundo(fd)

long Fseek(fd, offset, ptr)
long offset;

int Fputc(fd, c)

long Fread(fd, addr, nbytes)
char *addr;

long nbytes;

int Fwrite(fd, addr, nbytes)
char *addr;

long nbytes;

int Fflush(fd)

void Ftie(ifd, ofd)
```

DESCRIPTION

These routines provide simple fast buffered I/O. The routines can be called in any order. I/O on different file descriptors is independent.

Finit initializes a buffer (whose type is *Fbuffer*) associated with the file descriptor *fd*. Any buffered input associated with *fd* will be lost. The buffer can be supplied by the user (it should be at least **sizeof(Fbuffer)** bytes) or if *buf* is **(char *)0**, *Finit* will use *malloc*(3). *Finit* must be called after a stretch of *non-* fio activity, such as *close*(2) or *seek*(2), between *fio* calls on the same file descriptor number; it is unnecessary, but harmless, before the first *fio* activity on a given file descriptor number.

Frdline reads a line from the file associated with the file descriptor The newline at the end of the line is replaced by a 0 byte. Lines longer than 4096 characters will have characters deleted. *Frdline* returns a pointer to the start of the line or on end of file or read error. The macro *FIOLINELEN* returns the length (not including the 0 byte) of the most recent line returned by *Frdline*. The value is undefined after a call to any other *fio* routine.

Fgetc returns the next character from the file descriptor *fd*, or a negative value at end of file.

Fread reads *nbytes* of data from the file descriptor *fd* into memory starting at *addr*. The number of bytes read is returned on success and a negative value is returned if a read error occurred.

Fseek applies *lseek*(2) to *fd* taking buffering into account. It returns the new file offset. The macro *FIOSEEK* returns the file offset of the next character to be processed.

Fundo makes the characters returned by the last call to *Frdline* or *Fgetc* available for reading again. There is only one level of undo.

Fputc outputs the low order 8 bits of *c* on the file associated with file descriptor *fd*. If this causes a *write*(2) to occur and there is an error, a negative value is returned. Otherwise, zero is returned.

Fprint is a buffered interface to *print*(3). If this causes a *write*(2) to occur and there is an error, a negative value is returned. Otherwise, the number of chars output is returned.

Fwrite outputs *nbytes* bytes of data starting at *addr* to the file associated with file descriptor *fd*. If this causes a *write*(2) to occur and there is an error, a negative value is returned. Otherwise, the number of bytes written is returned.

Fflush causes any buffered output associated with *fd* to be written; it must precede a call of *close* on *fd*. The return is as for *Fputc*.

Ftie links together two file descriptors such that any *fio*-initiated *read*(2) on *ifd* causes a *Fflush* of *ofd* (if it has been initialised). It is appropriate for most programs used as filters to do **Ftie(0,1)**.

SEE ALSO

stdio(3), *print*(3)

DIAGNOSTICS

Fio routines that return an **int** yield **-1** if *fd* is not the descriptor of an open file or if the operation is inapplicable to *fd*.

BUGS

The data returned by *Frdline* may be overwritten by calls to any other *fio* routine.
Fgetc is much slower than access through a pointer returned by *Frdline*.
There is no *scanf* equivalent.

Formatted I/O in C++

Mark Rafter
..!mcvax!warwick!rafter
Computer Science Department
Warwick University
Coventry
England

ABSTRACT

The fmtio library extends C++ stream I/O to include formatted I/O in the style of stdio. This extension is layered on top of stream I/O, and only requires minor changes to **<stream.h>**. The key traits of the original stream I/O system, namely extensibility and type-security, are retained. An example of its use is:

```
cout[ "log of %d is:%9f\n" ] << 5 << log(5);
```

which prints

```
log of 5 is: 1.609438
```

The fmtio library is presented as a suitable framework in which to conduct further experiments with formatted I/O systems. The methods used in the library are sketched, and its overall structure outlined. An example is given of how to equip a datatype with formatted I/O by interfacing it to the fmtio library.

1. Introduction

The C++ stream I/O system is a great approach to I/O — it provides the programmer with a mechanism that is uniform, extensible and type-secure [1]. Stream I/O stops short of providing acceptable formatted I/O — its formatted output functions, **form**, **dec**, **hex** and **oct**, are clearly a stop-gap measure [2]. In particular, **form**, being a lightly disguised form of **sprintf**, delivers all of the problems that we associate with **sprintf** and its stdio cousins.

It was not clear to me whether formatted I/O had been omitted from the stream I/O library because of a mismatch between the formatted I/O idiom and the expressiveness of C++, or whether it was a routine piece of work that just hadn't got done. After some experimentation, I came to the conclusion that the answer lay somewhere between these two extremes.

Extensible, type-secure formatted I/O systems can be built in C++ — moreover the operator idiom of stream I/O, and the format-specifier idiom from stdio can both be retained. An example of such a library, fmtio, is described here. However, the methods used in its construction probably don't count as completely routine.

Is specifying features such as field-width, radix, justification and padding important to people, and if so why? The formatting facilities of stdio are breeding grounds for bugs, and this makes the real cost of using them quite high. The fact that people are willing to pay this cost might be taken as evidence that stdio's formatting facilities are valued. However, it is not the whole story.

The interface to stdio's formatting facilities can be made much safer. In C++ we could define overloaded **print** and **read** functions which would call **printf** or **scanf** in a manner appropriate to their argument types. This would be type-secure. It is even possible provide a safe interface to stdio in C, although then there would be an ugly proliferation of function names, **read_int**, **read_double**, **read_complex** etc. .

Why do we not see safer interfaces to the stdio facilities more widely used? It is not just the formatting facilities of stdio that people value, it is the notational advantages of the formatted I/O idiom, in particular:

- The format-specifier notation is compact.
- The formatting information is gathered together in one place, and the data objects are gathered together in another.
- The format-specifier notation is simple and well-suited to routine tasks.

In effect, the formatted I/O idiom provides a little language [3] for describing I/O.

In developing fmtio, notational convenience was considered as important as both extensibility and type-security. Notational inconvenience was considered grounds for rejecting a design. Consequently, the simple to implement approach based on overloaded functions was rejected as too cumbersome.

Although the fmtio library achieves its aims — the demonstration of a convenient, extensible, type-secure formatted I/O library for C++ — it seems premature to recommend that fmtio be adopted as a standard. Something much better than stdio-like facilities should be possible — more experimentation and further work is called for. What the fmtio library does offer is a nice framework for building formatted I/O systems. This paper sketches the methods used in the fmtio library and outlines its overall structure. Only formatted output is discussed; formatted input uses the exactly the same mechanisms. The main ideas underlying fmtio are:

- The COOL assumption
- Control flow based on the COOL assumption
- Making C++ look COOL

Some of these ideas may have applications in other areas.

Throughout this paper the following program fragment will act as an example of the use of fmtio library:

```
cout["log of %d is:%9f\n"] << 5 << log(5);
```

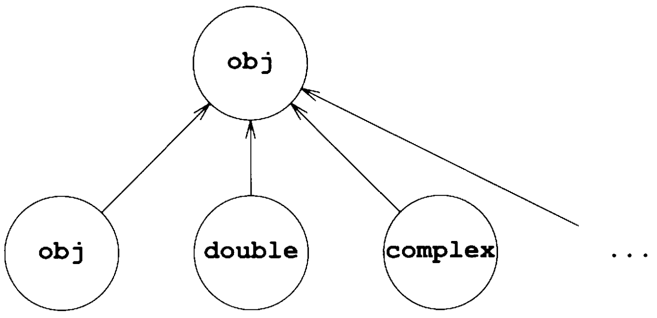
When executed, our example prints:

```
log of 5 is:  1.609438
```

The **%d** controls the format of the integer **5**, and the **%9f** controls the format of the double **log(5)**.

2. The COOL Assumption

The methods used in the construction of the fmtio library take their simplest form in a Completely Object-Oriented Language (COOL), i.e. one in which there is a type **obj** from which all other types are derived.



Of course, C++ is not such a language; but, in the interests of the explanation, we *temporarily* adopt the fiction that it is. In fact we go further than this and assume that the language has all the features that will make the implementation of fmtio easy. These are:

- All types are derived from the type **obj**.
- There is automatic coercion from any type to the base type **obj**.
- The base type **obj** is equipped with virtual **read** and **print** functions.

Each type is has its own redefinition of **read** and **print** that handle the low-level details of formatted I/O for this type, and this type only. For any given datatype, the provision of these functions is the responsibility of the datatype designer.

Later we will show how to approximate this ideal situation in real C++. For now, we continue with our fiction and give the declaration of the **obj** datatype:

```
struct obj
{
        virtual void  read( ostream & strm, char * fmt );
        virtual void print( istream & strm, char * fmt );
};
```

We illustrate the use of the virtual functions by reference to our example. Its execution should (somehow) result in the following two calls:

```
          int(5).print( cout, "%d" );

double( log(5) ).print( cout, "%9f" );
```

The way in which these functions are provided with their arguments, and how they are called, is the subject of the next section.

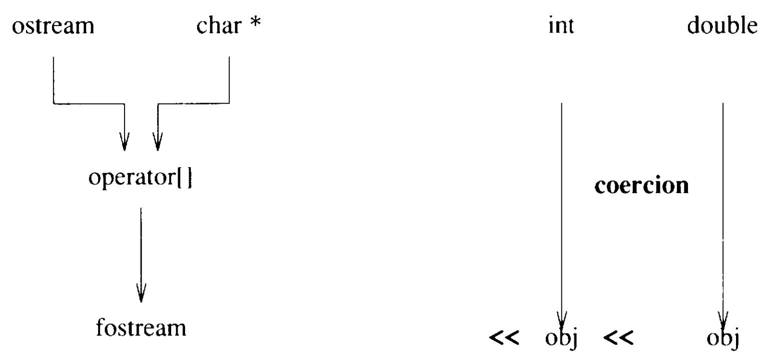
3. Control Flow Using the COOL Assumption

The fmtio library is almost entirely concerned with providing a control flow framework for the orderly execution of formatted I/O expressions. This framework is built with three main artifacts:

- The types **fostream** and **fistream**. These are the formatted stream types. Objects of these types contain information used in the execution of formatted I/O expressions.
- The index operators. These take an unformatted stream and a format-specifier (**char ***). The operators bind them together to give a formatted stream object.
- The formatted I/O operators. These differ substantially from their unformatted counterparts. There is only ever a *single* output operator and a *single* input operator. These are provided as part of the fmtio library. Both operators take a right-hand operand of type **obj &**. This, and the assumed automatic coercion of all types to the base type **obj**, allows the two formatted I/O operators to be applied to objects of any type.

The relationships between the above can be illustrated with our example:

```
cout["log of %d is:%9f\n"]    <<  5  <<  log(5);
```

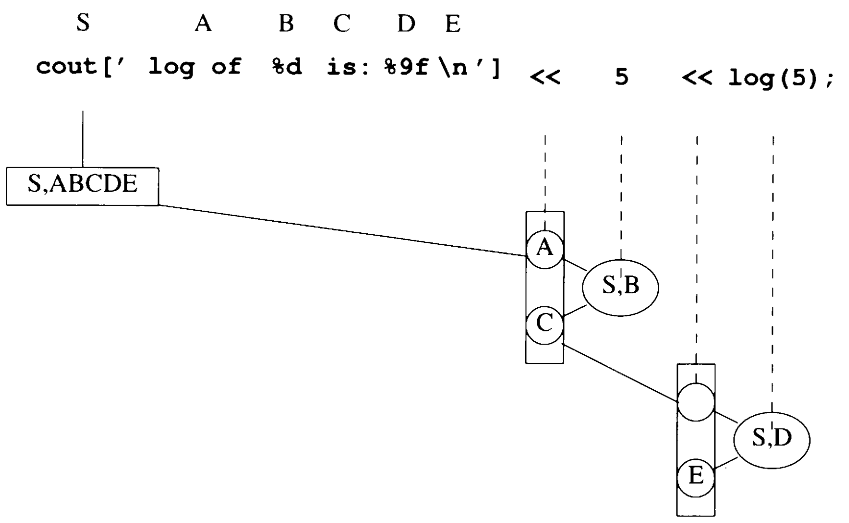


The execution of this formatted I/O expression starts when the index operator binds the unformatted stream **cout** and the format-specifier together creating a new formatted-stream object. Then control passes from left to right through each of the formatted I/O operators in turn, just as in the stream I/O system. As control progresses through the operators they deal with successive parts of the format-specifier. The operators record this progress in the formatted-stream object that was created by the index operator. The job of an I/O operator is in three parts:

- Process any plain text in the format-specifier up to the next format control character.
- Make a copy of the format control part of the format-specifier. Call the virtual **read** or **print** function for the right-hand operand. Supply the unformatted stream to be used and the copy of part of format-specifier as the function's arguments.

- Process any plain text remaining in the format-specifier.

We can trace the flow of control involved in executing our example. This shows which operators and functions deal with the various parts of the format specifier, and what they do with them.



Control flow starts with the index operator and passes to the left-hand output operator, which outputs the leading plain text **"log of "**. The left-hand output operator calls **int::print** with **cout** and **"%d"** as arguments. The function **int::print** outputs **"5"** on **cout** and returns to the left-hand output operator, which then outputs the trailing plain text **" is:"**. Control flow then passes to the right-hand output operator, which has no leading plain text to deal with. The right-hand output operator then calls **double::print** with **cout** and **"%9f"** as arguments. The function **int::double** outputs **" 1.609438"** on **cout** and returns to the right-hand output operator, which then outputs the trailing plain text **"\n"**. At this point the formatted I/O expression is complete.

The operators of the fmtio library are entirely concerned with imposing control flow and providing controlled access to parts of the format-specifier. All of the details of the actual formatted I/O operations have been delegated to the virtual **read** and **print** functions associated with the various datatypes. Consequently the only formatted I/O modules that are linked into a program are the ones it uses. This improves on the stdio situation where all or none of the formatting modules are linked in.

Because the formatting information in the format-specifier is decoupled from the data objects to which it refers, it is possible for the two to be incompatible. This situation does not pose a type-security threat to fmtio in the same way that it does to **form**, **printf** and **scanf**. In fmtio, the formatting algorithm is chosen on the basis of the type of the object concerned, not on the basis of data in the format-specifier. The issue of incorrect data in a format-specifier should not be ignored; just as unformatted input operators should cope with incorrect data in an (external) data-stream, formatted I/O operators should *gracefully* handle incorrect data in an (internal) format-specifier.

4. Making C++ look COOL

To adapt the above scheme to C++ we must simulate a unified type hierarchy and its coercion mechanism. We don't attempt to do this in complete generality — that would be too hard — we need only simulate the unified type hierarchy for formatted I/O expressions. To do this:

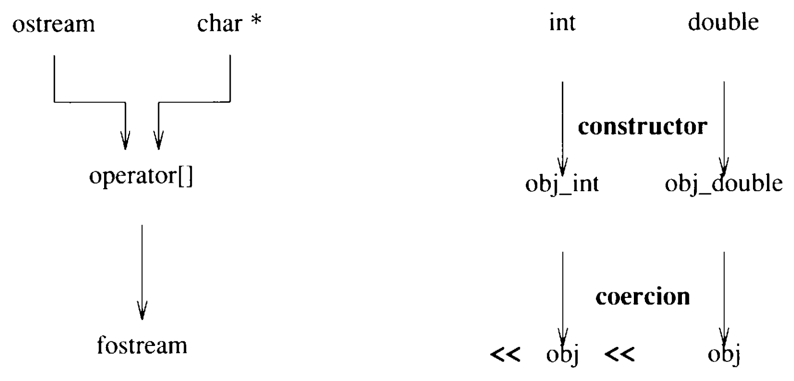
- Take as the base of our hierarchy the type **obj**, declared in exactly the form shown above.
- The **obj** hierarchy will be a hierarchy of container classes. The objects in this hierarchy will refer to the right-hand operands in formatted I/O expressions. Each C++ type, **T**, is injected into the **obj** hierarchy by deriving a type **obj_T** from the type **obj**.

We would like to hide all of this mechanism from the casual user of the fmtio library by exploiting the C++ implicit type conversion rules. As a (flawed) first attempt:

- Each of the container classes, **obj_T**, is equipped with a constructor taking an argument of type **T**. We want this constructor to automatically inject an object of type **T** into the **obj** hierarchy when the object occurs as the right-hand operand of a formatted I/O operator.

This realisation of our formatted I/O expressions is a slight variant of our previous scheme:

```
cout["log of %d is:%9f\n"]    <<  5  <<  log(5);
```



In fact this scheme must be modified slightly — the C++ implicit type conversion mechanism is not sufficiently strong. The constructor for **`obj_T`** will automatically convert a **`T`** into an **`obj_T`**, but only in a context that demands a value of type **`obj_T`**. In formatted I/O expressions the "generic" I/O operators demand right-hand operands of type **`obj`**, not **`obj_T`**, so the constructor is not implicitly applied. This is the case even though an implicit coercion from **`obj_T`** to **`obj`** is allowed. The two stage type conversion of **`T`** to **`obj_T`** and of **`obj_T`** to **`obj`** needs to be supplied more explicitly. To do this:

- Each container class, **`obj_T`**, comes together with a formatted I/O operator that takes a right-hand operand of type **`T`**. The body of this operator does no more than state that the right-hand operand must have the conversions **`T`** to **`obj_T`** and **`obj_T`** to **`obj`** applied before the "generic" I/O operator is used.

Although our scheme for COOL simulation in C++ may sound a little complicated, it is quite simple from the point of view of a datatype designer. Apart from the datatype-specific **`read`** and **`print`** functions a simple class declaration and an operator definition is all that is required to equip a datatype with formatted I/O. This class declaration would be unnecessary if type hierarchies could be suitably parameterised; alternatively the production of the class declaration could be automated with a suitable cpp macro — the author chooses not to do this.

As an example, here are declarations that are necessary to equip the type **`complex`** with formatted I/O:

```
struct  obj_complex : obj
{
        complex & datum;
        obj_complex( complex & d ): datum(d) {}
        virtual void  read( ostream & strm, char * fmt );
        virtual void print( istream & strm, char * fmt );
};

fostream & operator<<( fostream & fstrm, complex & z)
{
        return fstrm << obj( obj_complex( z ) );
}
```

A quick and dirty version of **`complex::read`** and **`complex::print`** can be built with stream I/O or stdio library functions — this is not very interesting, and so is omitted here.

5. Variations

The version of the fmtio library described here has the virtual **`read`** and **`print`** functions receiving their part of a format-specification via an argument that has type **`char *`**. The author has experimented with several variants of this mechanism, the most flexible of which is to make this argument an unformatted input stream [4]. The details of a format specification can then be read using unformatted, or even formatted, I/O operators. This makes handling field-widths etc. very easy.

Formatted I/O systems structured in the way we have described enjoy the characteristic that only the modules actually used in a program will be incorporated into it by the link-editor. Thus, for example, modules for floating-point formatted I/O will be omitted if only integer formatted I/O is used.

In **fmtio** there is only a single **obj** hierarchy; because of this both input and output operations for a data type will be incorporated even though only one of them is used. An alternative approach is to have two separate container hierarchies, **oobj** for output, and **iobj** for input. The virtual **print** functions and the output operators are defined for the **oobj** hierarchy. Similarly, virtual **read** functions and input operators are defined for the **iobj** hierarchy. Each datatype **T** must then be equipped with an **oobj_T** type for formatted output, and an **iobj_T** type for formatted input. With this arrangement, input and output modules will be incorporated independently.

6. Digression

Although not directly concerned with I/O issues, a possible weakness in C++ was turned up while investigating the above. This section deals with this — it is probably only of interest to C++ or OOP afficionados.

One of the strengths of C++ is that it provides support for object-oriented programming with all type-checking performed at compile-time. It has been claimed that this removes the run-time phenomenon of objects being asked to execute a method for which they don't possess a suitable definition. Such a claim is too strong.

It is not uncommon to declare a base type, equipped with public virtual functions, for the sole purpose of defining the common characteristics of a set of related types. The related types will be derived from the base type and they must redefine all the virtual functions declared in the base. This is what we did with our **obj** type and the other **obj_T** types derived from it.

In such a situation both the base type and the definitions of the virtual function found therein may be unwanted artifacts. No object of such a base type should ever be created; the base class definitions of the virtual functions should never be executed (assume they print out ''cannot happen'' and then dump core).

C++ gives us a mechanism for ensuring that a base type object can never be created — we make the constructors protected. Ensuring that the base class definitions are never executed is not possible because we cannot enforce the requirement that the virtual functions be redefined in all derived classes. The compiler is unable to complain if we erroneously forget to redefine one of these functions. For example, if we forget to declare the **print** function in the class **obj_complex**, the base class definition **obj::print** will be executed when formatted output of complex numbers is used. So we get ''cannot happen'' printed and a core dump — i.e. a message saying that an object has been asked to execute a method for which it possess no suitable definition — precisely the phenomenon that was supposed to be banished.

Is this a problem with the C++ language definition or with its current implementation? Strictly speaking it is the implementation that is defective. If the virtual functions were declared in the base class but not defined, then it would be illegal to create an object of a derived class, if that class were not to redefine all the virtual functions of the base class. There are two obstacles to this. Firstly, the compiler will need to do an analysis of the constructors for base types when compiling constructors for a derived type. Secondly, co-operation from the link editor may be demanded.

C++ needs a sophisticated separate compilation system, but the current implementation attempts to use a minimal one. This has been done for reasons of portability — it is common to have no choice about which link editor is to be used. Thus the the second obstacle may prove to be the greater.

7. Further Work

Layering the fmtio library on top of the stream I/O library is interesting, but has drawbacks. For example, it is easy to mix formatted and unformatted I/O operators in a single I/O expression but this doesn't work well. A version of the fmtio system that was integrated with the stream I/O library would solve this problem.

At the mundane level of the format-specifier syntax, something much better than the stdio style should be possible. For example, a syntax that defined, in a simple but flexible way, how much of the text following a `'%'` character should given to a **read** or **print** function would be a big help.

At a deeper level, we should probably examine more powerful ways of structuring streams of data; Rob Pike's use of structural regular expressions [5] looks like an interesting avenue.

8. Summary

The development of the fmtio library has demonstrated that a convenient, extensible and type-secure formatted I/O library with the best features of both the stdio and stream I/O libraries can be built in C++. More important than the library itself are the methods used in its construction and its overall architecture — its framework. What is needed now is further experimentation to see what facilities we want to place within this framework.

9. Acknowledgements

I would like to thank Bjarne Stroustrup for drawing my attention to an error in an earlier version of this paper [6].

10. References

1 Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, New Jersey, 1986.

2 Bjarne Stroustrup, *An Extensible I/O Facility for C++*, pp57-70, USENIX Portland 1985 Summer Conference Proceedings.

3 Jon Bentley, *Little Languages — Programming Pearls Column*, Comm. Assoc. Comp. Mach, pp711-721, Oct 1986.

4 Mark Rafter, *Formatted Streams: Extensible Formatted for C++ I/O Using Object-Oriented Programming.*, Research Report 107, Department of Computer Science, Warwick University, 1987.

5 Rob Pike, *A Tutorial for the sam Command Language*, Computing Science Technical Report No. 129, AT&T Bell Laboratories, Murray Hill, New Jersey, 1987.

6 Mark Rafter, *Extending C++ Stream I/O to Include Formats*, Proceedings of the USENIX 1987 C++ Workshop, Santa Fe.

Evolution of the SunOS Programming Environment

Robert A. Gingell

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043 USA

ABSTRACT

Recent changes to Sun's implementation of the UNIX operating system (SunOS) have provided new functionality, primarily file mapping and shared libraries. These capabilities, and the mechanisms used to build them, have made significant changes to the programming environment the system offers. Assimilating these new facilities presents many opportunities and challenges to the application programmer, and these are explored in this paper.

The new mechanisms also provide the application programmer with a flexibility comparable to that previously reserved for the operating system developer. Much of this flexibility is based on mechanisms for dynamic linking that support interposition. The future developments and ramifications of these mechanisms, as well as other areas for similar system refinements, are also explored.

1. Introduction

Several new capabilities have been added to Sun's implementation of the UNIX operating system (SunOS), most notably file-mapping and shared libraries. These capabilities not only introduce important new facilities, they also present new issues with which a programmer must be concerned, while at the same time enabling new performance and functional opportunities, and sometimes even making new classes of applications possible.

The manner in which these new capabilities have been incorporated into the system is also of interest. They are often applications of some more primitive and general underlying facility. These facilities are not necessarily new in the sense of being ''original'', all that is new is their implementation in SunOS. Their utility lies in being general, fundamental abstractions. They are interesting to the system architect because they provide a basis for a compact and efficient system implementation. However, they may be of more interest to an application developer, because the ''system'' functions they provide are but one of many possible applications for them – application areas now open to more than just ''system programmers''.

An examination of these facilities involves not only a description of how to use and work with them, but must also include the architectural and implementation decisions made in deciding how the system provides a given capability. For us, these decisions are driven by a desire to find the fundamental abstractions common to a group of related problems, and then address those problems uniformly from those fundamentals.

We do not claim there is anything particularly remarkable about these activities. Some would properly claim that they simply represent a restatement of precepts of long-standing programming disciplines or the ''UNIX-philosophy.'' Where we believe we may be particularly successful is in our consistency in following the abstraction process and in a willingness to reimplement extant parts of a system to maintain architectural integrity. This approach leads to a particularly powerful system architecture, one that more easily lends itself to future evolution than if we had provided a specific capability as a closed system addition. It also transcends arbitrary software boundaries such as ''the kernel'', instead representing a systemic philosophy of providing abstractions through basic mechanisms that are applied to create (or perhaps recreate) a specific system capability. Further, this approach helps retain the ''essential character'' of the UNIX system, allowing us to extend it and at the same time provide a powerful and efficient implementation of standard interfaces and thus gain from the large and growing body of UNIX software.

The sections that follow will briefly describe the new facilities present in SunOS. For the most part, these are more completely described elsewhere, and the discussion provided here is simply for background. Also examined will be the application of these facilities to recreate portions of the system based on new fundamentals, and the issues and potential difficulties the new facilities can present to application developers. Then our experiences in applying these facilities will be described, as well as a perspective on how they will evolve and the opportunities they present for application development. In addition, future opportunities for the evolution of the system will be explored.

2. New Virtual Memory System

The demands to support different forms of shared memory, mapped file access, shared libraries, and practical demands of increased system portability have led us to the development of a new Virtual Memory (VM) system for SunOS. The new system unifies all the system's operations on memory (including the ``memory'' in files) around the single notion of file mapping. The following summary covers the system's general concepts and how they were used to reimplement existing operations on memory. A more complete examination of the systems architecture and operations can be found in [GING 87a] and a detailed treatment of its implementation structure found in [MORA 88]. Conceptually, the system owes much to MULTICS [ORGA 72] and TENEX [MURP 72].

2.1. General Concepts

The new VM system provides a page-based facility in which the fundamental concept is file-mapping. The system's *virtual memory* consists of all its available physical memory resources, including local and remote file systems, pools of unnamed memory (*swap space*), and other random access memory devices. Named objects in the virtual memory are referenced through the UNIX file system. Previous SunOS work on file system interfaces permits many different implementations of file objects that are manipulated through an abstraction of the original UNIX *inode*, called a *vnode* [KLEI 86]. The object manager for a *vnode* is called a *Virtual File System* (VFS), and is itself an abstraction of the services required to implement a file system.

A process's *address space* is defined by mappings onto objects in the virtual memory. The mappings are constrained to be sized and aligned according to the page boundaries of the system on which the process is executing. Each page in an address space is independently mappable (or not), and thus the programmer may treat the address space as a simple vector of pages. A given process page may map to only one object, although a given object address may be the target of many process mappings. An important characteristic of a mapping is that the object to which the mapping is made need not be affected by the mere *existence* of the mapping. The implications of this are that it cannot, in general, be expected that an object has an ``awareness'' of having been mapped.[1] Establishing a mapping to an object simply provides the *potential* for a process to access or change the object's contents.

The establishment of mappings provides an *access method* that renders an object directly addressable within an address space. Unlike the access methods provided by *read* and *write* that require an application to operate only on a copy of object data (i.e., a program buffer), this method eliminates the inefficiency of copying while permitting the object to retain its identity during the access operation. The ability to directly access an object and have it retain its identity over the course of the access is unique to this access method, and promotes sharing of common code and data.

The VM system consists of programming that operates as a cache manager for data in the virtual memory. The physical resources for the cache are the processor's primary memory. References to VM objects result in either an access to a ``cache entry'' or the fetching of data into the cache, possibly requiring removal of other data. The latter two functions are simply ``page-in'' and ``page-out''.

An important characteristic of the system is the balance between the responsibilities of the VM system as cache manager, and those of the VFS that obtains data for filling a cache entry and to which data is passed when flushing an entry. This balance permits different handling of requests based on the object manager, where the differences may reflect either semantic changes or performance enhancements, or both. For example, predictive operations such as the old function of ``read-ahead'' are supported by object managers that ``page-ahead'' in response to sequential accesses.

[1] It is not *prohibited* for an object to be aware of being mapped, it is simply not guaranteed that all objects *can* have such awareness.

2.2. Application in System Primitives

The most basic operation is that establishing a mapping between a process address and an object in the virtual memory. This operation is available through the *mmap* system call, fully specified in the system's architectural description and first described in [JOY 83]. It should be noted that the only memory that a process can address is that to which a mapping has been established.

mmap provides for two primary "types" of mappings: one, called MAP_SHARED, creates a mapping that lets store operations change the mapped object (i.e., the result of the "write" is shared with all users of the object). The second type of mapping, MAP_PRIVATE, creates a mapping that makes the changes made by store operations private to the address space containing the mapping. MAP_PRIVATE is often referred to as *copy-on-write*, reflecting a common implementation technique of intercepting the first store to a page, copying the page, and redirecting the original store and successive references to the copy.

The mapping abstraction for accessing memory has been used in a reimplementation of several UNIX kernel operations. These include *exec*, *fork*, and *brk*. Perhaps surprisingly, these also include *read* and *write*. The reimplemented operations still exist as system calls to retain a compatible interface for old applications. However, from a memory management perspective there is little motivation to implement them this way. The common uses of the *read*, *write*, and *brk* system calls in particular can easily be implemented in application code using *mmap*. And all of the functions of *exec* save the implied "overlay-and-jump" function could similarly be implemented outside of the kernel, a fact used in the development of shared libraries.

2.2.1. *exec*

exec overlays a process's address space with a new program to be executed by performing an internal version of *mmap* to the file containing the program.[2] The process's stack and uninitialized data areas are mapped to unnamed, zero-initialized storage. The mappings *exec* establishes are all the MAP_PRIVATE type.

The use of MAP_PRIVATE mappings simplifies the system call *ptrace*. Formerly, *ptrace* would refuse to deposit breakpoints or otherwise write on the text of a program executing in more than one process. With the changes to *exec*, this restriction has been removed: *ptrace* does its work by setting a text page writable, depositing its breakpoint, and restoring the write-protection. Because the page where the breakpoint is deposited is mapped MAP_PRIVATE, when *ptrace* stores into it, the store will really access a copy.

2.2.2. *fork*

Perhaps the most interesting application of the new facilities is in *fork*. In the new system, *fork* has been redefined. Formerly it would copy the address space of the parent to build the child. It now merely copies the mappings describing the address space. If a mapping was MAP_PRIVATE in the parent, it is MAP_PRIVATE in the child, and neither sees changes made by the other. Copies of address space data are made only if necessary, and then only a page at a time. Since most *fork*s are followed immediately by an *exec*, this avoids wasting substantial effort in making copies that are never used.

If a parent mapping is MAP_SHARED, then the corresponding mapping in the child will also be MAP_SHARED. This provides the capability for parent and child to share memory after a *fork*, something that was not previously possible in Berkeley-based UNIX systems [JOY 83].

The new *fork* illustrates that it is possible to reimplement and even redefine standard operations while both maintaining compatibility and improving system functionality and performance. Although the definition of *fork* has changed to take advantage of a more general underlying memory management mechanism, its function to programs not using the new features of the system is completely compatible. And those programs that take advantage of functions not previously available (e.g., MAP_SHARED mappings of files) have a more powerful facility with which to work.

2.2.3. *read* and *write*

When mappable objects are accessed through the *read* or *write* system calls, the kernel performs the internal equivalent of an *mmap* to the kernel's address space to gain access to the file. This is followed by the appropriate *copyin* or *copyout* operation to drain or fill the caller's buffer. The changes to *read* and

[2] The code and data in this file must be in the (default) "demand-page" format. Executables in other formats are handled by copying them to a paged format in mapped unnamed storage.

write unify the system's I/O functions on memory objects around the notions of mapping. Mapping access to files has not been some extra wart added on top of the system. Instead, the right abstraction has been identified and then used to ensure that comparable system functions could be expressed in terms of it – thereby simplifying the system and its implementation.

2.2.4. "Segments"

The new system provides a programmer with an address space that can be viewed as a simple vector of pages. The old notions of "text", "data", and "stack" segments no longer have any real meaning to the *implementation* of the system. The system as a whole retains the notions, but they are conventions imposed by the language tools, rather than a fundamental system-imposed constraint. For compatibility purposes, *exec* still establishes a range of memory to use as a heap, another for a stack, and *brk* adds or removes mappings to manage a "data segment". However, there is nothing to stop a program from having multiple non-contiguous heaps, or multiple text and data segments. The latter actually occurs when running programs constructed with shared libraries.

3. Dynamic Linking and Shared Libraries

The *text* of a program consists of some body of code that implements the function for which the program was written and also of code copied from libraries. Although UNIX has long supported sharing code among processes running the same program, the fact that nearly every program makes use of routines such as *printf* means that at any given time there are as many copies of these routines competing for system resources as there are different programs. As the body of UNIX programs grows, so does the percentage of system resources devoted to these copies. The notion of "shared libraries" attempts to extend the benefits of code sharing to processes executing *different* programs, by sharing the libraries common to them.

The following sections summarize Sun's approach to providing a shared library facility; a more complete treatment can be found in [GING 87b].

3.1. General Concepts

Shared libraries in SunOS are provided through the application of other mechanisms. These are:

- a revised system link editor (*ld*) that supports dynamic loading and binding;
- use of the file mapping facilities to introduce an object (i.e., a file containing a shared library) to an address space; and
- compiler changes to generate *position-independent* code (PIC).

It should be noted that the only one of these mechanisms provided by the UNIX kernel is file-mapping. This is simply the *mmap* function. There is *no* kernel support specific to the support of shared libraries.

3.1.1. *ld*

Many of the functions relating to shared libraries are embedded in the link-editor, *ld*. Conceptually, *ld* has been transformed from a just a batch utility that combines object files to a more persistent facility, available to perform link-editing functions at various times over the life of program. Previously *ld* built all programs *statically* – executable (*a.out*) files contained complete programs, with all code and data bound and relocated in a single batch operation. The new *ld* will build "incomplete" *a.out* files, deferring the incorporation (and binding) of certain object files until some later time (generally program execution). Still other bindings (procedure calls) are deferred until the object is first referenced.

The object files on which *ld* defers link editing are added to the address space at execution time using the system file mapping facilities to address and thus share these objects directly. These "shared object" (*.so*) files are simply executable files (demand page format) lacking an entry point.

Dynamic link editing is still the same operation as static link editing, all that has changed is the *time* at which it occurs. But, what a linker such as *ld* does is *edit*: it changes a program reference to resolve it to something a processor can execute directly. To change something involves writing on it, and to write on it means that it can no longer be shared. What has really been built with these facilities is not so much "shared libraries", but "*dynamic* libraries". "Shared" becomes a property of code that can be added to a program directly without any change, rather than a functional characteristic.

3.1.2. PIC

To make code more sharable, the C compiler was enhanced with an option to generate position-independent code (PIC). PIC does not require relocation to be incorporated into an address space, and is thus inherently sharable. This is accomplished by generating references to static storage as indirections through a *linkage table*. When *ld* link edits a PIC module, it builds this linkage table, initializing it with pointers that will eventually themselves be relocated.[3] The PIC references to the linkage table itself are relocated by *ld*, leaving pure ready-to-execute code behind.

Generally, *.so* files are built as PIC. At execution time all that need be relocated from these executables are their linkage tables. These are usually small relative to the entire object file. However, the use of PIC to avoid dynamic link editing operations is simply a *performance* optimization, *not* a functional one. While it will be shown that this optimization is important for effective system operation, not constraining the mechanisms to enforce sharing provides an important flexibility advantage.

3.2. Use of Shared Libraries

In addition to supporting static and dynamic linking, *ld*'s conventions regarding the interpretation of its `-l` option were augmented. `-l` is the shorthand reference for a library name, implying both a search path and a name format. For instance, a command line such as

```
% cc -o ... -lx ...
```

formerly meant search for library `x` in a file named `libx.a`, located in one of several directories, specified explicitly or implicitly. The new *ld* performs this function as well, but expands on it to allow the library to have a different name, specifically `libx.so`. If *ld* is enabled to perform dynamic linking (now the default), it will search for either the *.a* or *.so*, preferring the *.so* form if both are present. A ``shared library'' is thus simply a *.so* file containing the objects comprising the library, named according to the format of library file names, and placed in one of the standard directories.

Since these mechanisms are applied after a program has been compiled or assembled, it follows that building a program with shared libraries involves no change to either the program nor the manner in which it is built. All that is required is to install a form of the library suitable for dynamic linking. In our system, for instance, there are both `libc.a` and `libc.so`. The *.so* form is almost universally used.

3.3. Version Control

To handle the independent evolution of shared libraries and the programs that use them, a version control scheme has been established. The *.so* files used as shared libraries really have a more complex name, involving a suffix that describes the version of the library contained in the file. *Interface version* ``2'' of the C library, in its third compatible *revision* would be placed in a *.so* having the name `libc.so.2.3`. The suffix may be an arbitrary string of numbers in Dewey-decimal format, although only the first two components are significant to the operation of the link editors at present.

The first component of the string (often called the ``major version number'') describes the library's interface, and the second component (``minor version number'') documents implementation revisions to that interface. When an application is linked by *ld*, the interface numbers of each of the library *.so* files *ld* processed are recorded in the dynamic linking information retained in the resulting executable. At execution time, this information is used by the dynamic link editor to determine the ``best'' library to use in an environment that may contain multiple versions of a given library. The rules followed are:

- **Interfaces identical**: the interface used at execution time must exactly match the version found at *ld*-time. If an exact match cannot be found, the dynamic load will fail.
- **Most recent revision**: in the presence of multiple revisions of a given interface, the one with the highest revision number will be used. A warning is issued if a revision appears to have been deleted since the application was built, although execution will continue.

4. Issues

As with most changes in technology, the developments in SunOS bring both benefits and challenges. The benefits are largely in programmer abstractions that provide improved performance, increased flexibility, or greater functionality. The challenges are often less clear, and pose subtle (and therefore insidious) issues

[3] *ld* really builds two ``linkage tables'', one for data references and the other for procedure calls. The procedure call table is (usually) loaded with code and more properly thought of as a ``jump table''.

that, if ignored, can trap the programmer trying to exercise a newly-available tool. Such issues include:

- **Illusions of Compatibility**. A pair of implementations are often considered "compatible" if they present the same "interface". However, interfaces generally only describe a narrow set of positive assertions about a facility: what a user *can* assume about its use. It is rare, however, for an interface to specify what its users may *not* assume. Further, interface descriptions are generally devoid of non-functional issues such as performance.
- **Inflation of Responsibility**. More powerful programming tools, perhaps paradoxically, often bring with them the obligation to use increased thought and skill in their application. Correct and skilled use brings increased benefits, careless use merely brings more spectacular disasters.
- **Threats to Conventional Wisdom**. Programming habits are acquired through a series of experiences that train the programmer in "what works". However, new tools and facilities, particularly those that generalize or refine an abstraction or the environment in which they are applied, threaten those habits. A practice once considered acceptable may no longer be adequate because the problems to which it is applied may have scaled beyond its applicability.

4.1. Working Set Size

One of the significant side-effects of the availability of shared libraries has been an increase in working set size [DENN 72]. This effect was expected, and has appeared in other implementations of shared libraries [ARNO 86]. The effect can be explained by the nature of a shared library: rather than including in the address space of a program only the library functions actually required during its execution, the *entire* library is provided. The (usually proper) subset of the library really needed is likely to be distributed over more pages, and thus the memory demand imposed by the program is greater.

Although more pages are required, as long as the pages are shared among multiple processes the incremental cost is decreased. This emphasizes the need to minimize the amount of *private* working set size, and suggests that the priorities for minimizing size are to first:

- reduce the percentage of the working set that involves physical memory unique to this process; and then
- reduce the shared physical memory requirements.

Experience has shown that the system is effective at handling pages that are truly shared, in that they are rarely removed from memory and thus impose little per-access cost. However, the per-process private pages must compete for the remaining physical memory resources with a comparable set of pages from other processes.

Although it has always been important for programs to impose a reasonable resource demand on the underlying system, the "working set" inflation created by shared libraries makes this a consideration not just of the application builder, but of the library builder as well. Although the functional requirements of the library have not changed (the interfaces are *compatible*), the performance side effects of this library "program" are not compatible with previous engineering considerations. It is, for instance, perfectly acceptable to ignore these issues and create a library that *works*, however the global impact of something like a poorly shared C library is potentially devastating.

4.2. Improving Sharing

The programmer can use several tools to increase the degree of sharing in a program or library and thus address the issues of working set size. Although these tools have always been available, the programmer may not have been sufficiently motivated to use them. In some (more insidious) cases, the changes to the system have caused the tools to be "broken" or otherwise changed in some non-functional way. For instance, the SunOS C compiler has supported an option (`-R`) to enter initialized data in the text segment of an object file. When programs were linked with archive libraries containing such objects, the read-only data was shared among all the users of a program. However, with position-independent shared libraries, the use of an option such as `-R` is no longer as simple: a PIC module built with `-R` and containing an initialized array of character pointers will actually *worsen* the sharing of code, since the pointers in the initialized data will require relocation when the object is actually added to the program at execution.[4]

[4] An alternative to `-R` is to have a C compiler that supports the `const` storage class: when generating PIC for a `const` data definition the compiler could issue the pointers and invariant data under separate relocation counters.

While it is important to move the truly invariant data to sharable memory, it is also important to recognize what is really a piece of invariant data.

Invariant data can also be obtained by recoding programs to use position-independent data (PID). Although a compiler can be changed to emit PIC, the position-dependence of static storage is a function of the program's algorithms. Clearly C is a language that favors a coding style using pointers, but often substantial efficiences can be gained if a data structure is accessed as PID, using relative addresses (array indices) to describe the desired address. The C treatment of array names as pointers makes this coding simple, if a little unnatural. Languages such as C++ [STRO 86] that contain support for overloaded operators such as ' [ ] ' can make this more aesthetically satisfying.

The use of PID is also important for databases that are expected to be accessed through mapping operations. Such databases might include, for instance, character font descriptions for a bit-map display, and be used by many processes simultaneously. Having the data be PID allows the client applications to structure their address space around their application, rather than the requirements imposed by a common piece of data.

Finally, private data storage requirements can sometimes be lessened through the use of dynamically allocated global storage. This can be accomplished by removing static declarations and changing their references to access lazily *malloc*ed data. In a body of code such as the C library, where the average program uses little of the entire library, this can represent a substantial space savings. Further, the data that is used is generally allocated contiguously with other used data, often on the same page of the *malloc* heap.

4.3. Interfaces and Configuration Management

The availability of a version control mechanism for dynamically linked objects places a requirement on the programmer to recognize and manage library interfaces. The programmer's responsibilities range from simply updating version numbers appropriately to designing the interface to be more suitable for dynamic linking. In addition, the ability to selectively use the dynamic linking facilities may require some decisions about program configuration management with respect to dynamic linking.

4.3.1. Interfaces

Although the management of library interfaces has always been important, before the availability of dynamic linking an erroneous or unanticipated incompatibility would not break *existing* programs. With dynamically loaded objects, such errors are possible. However, they are also generally easier to detect earlier in the development cycle. Because the new code can be easily inserted into almost every program in a system, it becomes easier to perform large-scale testing.

A problem with interface management is that the languages and tools most programmers use do not provide much support for it. Further, the interface is more diffuse than just the functions, arguments, and results that one usually associates with a library. It extends to the shape (and sometimes content) of data structures shared by, or defined in, both applications and libraries. For example, if the size of a `jmp_buf` (as defined in the file `setjmp.h`) changes, then this constitutes a C library interface change. If the interface did not change, the extant applications containing instances of `jmp_buf` structures might fail if run with a C library that expected the differently sized `jmp_buf`. If the size the library expects is greater than that built in to the program, then the application will most likely malfunction as the library routines overwrite the area reserved in the application.

4.3.2. Configuration Management

The provider of a given application (or set of applications) may not wish to expose itself to changes made to libraries on which the application depends. Yet, the application may consist of application-specific libraries that are dynamically linked to simplify maintenance. Further, when an application *does* fail, the problem of determining the environment in which the failure occurred is more complex. Not only must one determine exactly what version of an application failed, it is important to know what versions of shared objects were involved. We have found it important to develop tools such as the program *ldd* (*l*ist *d*ynamic *d*ependencies) that displays the shared objects used to execute a given application. We have also modified other tools, notably debuggers, to interpret a program's dynamic configuration.

The apparent complexity of the configuration management issues is not so much a problem as it is a manifestation of an opportunity: vendors and computing suppliers now have a vehicle that supports multiple interfaces simultaneously. It has become more practical to ship "field-replaceable" software units as libraries, because a mechanism exists that no longer requires an "all or nothing" form of program replacement.

Further, the specification of interfaces at a dynamic linking level encourages innovation. Programmers modifying or enhancing the UNIX kernel have benefited for years from the ability to replace the code behind an interface that was dynamically linked with applications. This flexibility has now been extended to more than just kernel developers – now virtually any programmer producing a general purpose facility has in the concept of a library much more than "a related collection of object files", there is an *interface* that exists independent of its implementation.

4.4. Foiled Assumptions: "Unexec"

Some facilities exist that are predicated on old conventions and practices. They presume a limited model in such a highly constrained manner that any attempt at generality foils them. An example is "unexec" utilities common in the building of programs that are "saved" in a memory-image initialized state. UNIX examples of such programs include TeX and some versions of the EMACS text editor.

These facilities were built on assumptions that the three segment memory model of a process was all there could ever be, something the use of shared libraries and dynamic binding invalidates. It is not that this function cannot be provided, simply that the considerations on which it is based need to be changed. It can be argued that "unexec" is simply the ability to save a program image, something that *ld* performs when finished link editing a program. This suggests that the dynamic abstraction of *ld* is incomplete, and that future work to complete it would include the capability for saving a program image.

5. Experiences and New Capabilities

The changes to the system brought about by the new VM and dynamic linking facilities, while creating sometime subtle changes in the programming environment that require increased programmer vigilance, have mostly brought about a more flexible environment in which it has become possible to vastly simplify or improve applications. In some cases, the changes have made it just possible to write a given application. In this section, we examine a few of these advantages to illustrate both their immediate value and their potential future impact.

5.1. Address Space Freedom

One of the goals of the new memory management facilities was to provide programmers with an address space that could be used flexibly. No longer does the operating system enforce any particular structure to the address space, nor does it impose any semantic requirement on any specific area of it. *Requirements* on address space structure have become *conventions* applied by language tools and utilities.

The resulting flexibility has simplified the implementation of shared libraries. A dynamically linked program consists of multiple text and data segments: one for each *a.out* file mapped into the program's address space. The notions of a "text" or "data" segments remain useful for describing the assembly of executable files, but no longer have a system interpretation. Without this generality, the mechanisms might have required specialized kernel support. We have often found that the imposition of a fixed semantic interpretation by a lower software layer eventually forms a barrier to flexibility.

Other uses of the unstructured address space have involved disjoint collections of data. Coroutine libraries creating multiple stacks now create them surrounded by "holes" in the address space that form *red zone* protection areas. Multiple storage heaps (either to improve locality or to isolate data for placement in stable storage) can also be easily established. The stable storage heap is simpler still: the stable storage is simply mapped into the address space for direct access.

5.2. Incremental Maintenance and Development

The dynamic linking of libraries permits easy incremental maintenance. A bug-fix to a library is easily incorporated into programs that use the library simply by installing the repaired version. This effect can be extended naturally to the program development process as well.

Developing a complex program involves many compile, link, and debug cycles. If the time taken to perform any of these operations can be shortened, programmer productivity will be improved. With the dynamic linking facilities in the new system, a crude form of incremental development is easily practiced. Consider a program consisting of many individual object files, such that the program is built with

```
% cc -o prog main.o ... many other .o files ...
```

Using shared objects, it is possible to build a "linking hierarchy" such that for any one edit and compilation of a single object file, only a few objects need be processed by *ld* rather than the entire set.

By way of illustration, assume a program built from 100 object files. For the purposes of this example, let each object file be named from `00.o` to `99.o`. A possible hierarchy might be to collect the ten *.o* files with the same leading digit into a single *.so* file, such as with:

```
% ld -o 0.so 0?.o
```

The program would then be built with:

```
% cc -o prog 0.so 1.so ... 9.so
```

that creates `prog` from 10 *.so*s dynamically linked at execution. If at some later time, a bug requires the recompilation of `23.o`, then only the modules comprising `2.so` (or just 10% of the total number of *.o* files) need be relinked to incorporate the change.

Of course, each time that `prog` executes, it incurs the cost of dynamic linking. However, this cost is often negligible when compared with static linking, since the dynamic linking process does not require that a new output file be built and written to the file system. Judicious grouping of *.o* files may provide further efficiencies.

Other groupings are possible of course. In the limit, each single object could be built as its own *.so* file. In practice, there is most likely some middle ground between the one-to-one extreme and complete relinking.

5.3. Application Performance

The access method provided by mapping has two fundamental performance advantages over its counterparts *read* and *write*:

1.) a buffer copy operation is avoided, reducing the demand for processing power; and

2.) not having to support access to a second block of memory to contain the same information reduces the demand on the system's memory resources.

This suggests that programs in which buffer copying represents a significant fraction of the execution time can benefit from optimizing out the copy with mapped accesses.

cat and *cp* have been reimplemented to incorporate this optimization, and use *mmap* rather than *read* when accessing a mappable file. This has had the dual advantage of removing both copy overhead and reducing the number of system calls. These programs map a large section of the file being read (one megabyte) and write it out in a single operation. A code fragment from a very simplified version of a *cat* that simply maps the entire file is:

```
...

int      fd;              /* file descriptor */
struct   stat sb;         /* file status */
caddr_t  cp;              /* file pointer */

...

/*
 * Take "fd" (opened for read), and map the entire length of
 * the file.  Write it to standard output with a single write.
 */
(void) fstat(fd, &sb);
cp = mmap(0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
if (cp != (caddr_t)-1)
        (void) write(1, cp, sb.st_size);

...
```

An interesting side effect of the use of mapping in these programs is that it provides a useful illustration of the "lazy evaluation" properties of the VM system. Directing their output to `/dev/null` will cause the program to take practically *no* time to execute, no matter how big the input file is. The reason for this is that *mmap* really performs no access on the data mapped, it merely describes to the system the potential for an access to occur. The driver for `/dev/null` on the other hand, never performs the access. The above program, when asked to do nothing, really does nothing!

5.4. Multithreaded I/O

If an application required the ability to perform multiple I/O operations in parallel, it had to be programmed using non-blocking I/O facilities and the *select* system call. Even with this, many I/O operations remained synchronous, since facilities such as the file systems did not support non-blocking I/O. With mapped files, the problems of initiating parallel I/O are even more acute, as I/O is only performed in response to a page fault, and such faults occur only on a single page at a time.

However, multiple processes can be created that are identically mapped (or have a significant overlap) in the construction of their address spaces. These processes can each perform their own I/O requests (or page faults) by dividing up the work between them. The maximum amount of parallelism is limited by the number of concurrent processes available.

This illustrates a crude example of multiple processes executing from a single (or heavily overlapped) set of memory objects. A more refined example is the notion of *lightweight processes* [KEPE 85], addressed in more detail further on. However, this example illustrates an important facet of the new system's architecture: the independence of address space contents from specific processes.

5.5. Global Performance Analysis

To support the development of the dynamic link editor, it became desirable to profile its execution. The granularity of the profiling process (generally at a line-clock rate), coupled with the short execution of the linker, required many samples. The problems of initializing and saving of profile buffers also presented a number of issues that were easily addressed with mapped files. A special version of the dynamic link editor was created that would map a specific file into its address space as part of its initialization. The kernel's profiling facilities were then directed to the mapped area. Other parts of the link editor accumulated other statistical information into the mapped area.

By mapping the file into the address space *shared*, all instances of the link editor in the system (nearly every process) contributed to the statistics collection in the shared file. At the end of a sample period of several hours, a significant set of information had been collected, one that almost certainly reflected an excellent description of how the link editor's time had been spent. The use of a mapped file obviated the need for the linker to engage in end-of-program activities to dump and merge the information with previous execution, and thus avoided any issues of saving the information from programs that did not end normally. Further, the information collected came from a large variety of programs, summed over the course of a long period of system execution. Although this example describes an implementation unique to the dynamic link editor, it suggests a general means of performing global instrumentation of a given piece of code, one that could benefit the construction of tools to improve the performance of programs on a global basis.

5.6. Interposition

A powerful aspect of the dynamic linking facilities is the ability to *interpose* targets for symbolic references. For example, consider the building of a program that uses a library, `libinterpose`, supplied as a *.so* file:

```
% cc -o prog prog.o -linterpose
```

This command invokes *ld*, and invisibly includes a reference to the C library after `libinterpose`. `libinterpose` defines entry points for *read* and *write* that in addition to performing the required system calls take statistics on the use of these system calls.

In this case, both `libinterpose` and `libc` define entry points for *read* and *write*, the latter being the standard entry points for the "stub" routines performing the respective system calls. In the presence of multiple definitions of a symbol, the link editor resolves the issue of which to select by using the ordering established when `prog` was linked: since `libinterpose` was specified first, its entry points are used for references to *read* and *write*. These entry points are used globally, so that *read* and *write* references made internal to the C library (such as from standard I/O routines) also use `libinterpose`.

Interposition offers the opportunity for programmers to provide "added value" to a "standard" system function by providing a new implementation of the function. In this respect, the act of interposing is the *control-flow* analog to the effect of UNIX I/O redirection on *data-flow*. An example of such a "value-added" feature is an instrumented *malloc* function, used to drive a graphical display termed a "mallometer". One approach to implementing this would be to replace *malloc* and build a new shared C library. While this would be more than sufficient, it presumes that it is possible to replace a single module in a shared library, which it is not. Further, the obligation to rebuild the library is unnecessarily cumbersome.

An easier approach is to ``insert'' a modified *malloc* into an already running program. This could be accomplished by building a *.so* out of just the modified *malloc* (say, `malloc.so`) and providing it to programs with a sequence such as:

```
% setenv LD_PRELOAD malloc.so
% prog
```

where LD_PRELOAD is interpreted by the dynamic link editor as a list of objects to be loaded before loading those requested by the program itself. The *malloc* defined by `malloc.so` would thus take precedence over that in the C library.

An even more flexible capability would be to allow a new *malloc* to simply ``jacket'' the ``real'' *malloc*, thereby not requiring any rewriting or reimplementation of the real function. The use of an interposing routine in this fashion can be viewed as the control-flow analog of a *filter* [RITC 74].

6. Trends and Future Developments

Development follows cyclical patterns: new developments bring new understanding that in turn brings further developments. The new facilities have brought with them both new capabilities and new requirements for their evolution. In areas such as dynamic linking, the application potential has barely been explored, as shared libraries represent but one potential use of the mechanisms. And, there are yet other areas of the system beyond memory and binding that can benefit from such evolution.

6.1. VM

Future developments of the VM system are likely to focus on the management policies of the system, such as the global page replacement algorithms and processor and memory scheduling. The likely outcomes of these developments include a page replacement policy supportive of a high degree of memory sharing, and an integrated process and memory scheduler. Support for program-directed performance functions such as ``advising'' the system as to expected process behavior (as in ``these pages expected to be needed soon''), or ``commanding'' the system to have some behavior (as in page locking), are expected to be created from these efforts.

6.2. A Global View of Binding

Many problems in operating systems and programming languages reduce to issues of ``binding''. For example, the VM system implements a per-reference address binding operation using hardware assists such as memory management units supplemented with software interpretation on exceptional conditions (page faults). The ``logical name'' the program uses is simply a process address, and the VM facilities translate the name to some physical storage address, a binding that occurs on each memory reference. The existing link-editing mechanisms provide a ``global symbol to process address'' binding. The dynamic linking facilities have changed this activity from one that is ``compiled'' to one that is ``interpretive''. Finally, many problems in distributed computing, such load balancing, server selection, and failure containment, are problems of binding a client to a specific instance of a server.

The binding mechanisms implementing shared libraries do not yet offer any direct functional interface to the programmer, whose interface is through command-line interaction with *ld*. When contrasted with the VM changes, which provided functions to programmers in the form of system calls such as *mmap*, the dynamic binding mechanisms have at present but one application: shared libraries. The mechanisms themselves have not been made directly available – though this reflects product and business priorities rather than an architectural limitation. The global use of shared libraries suffices to establish the architecture into which new capabilities and interfaces can be easily introduced.

The existence of an interpretive binding facility in the programming environment architecture presents many opportunities for innovation not just by UNIX system vendors, but also by independent parties and individual programmers. These opportunities begin with the availability of a generalized set of link-editing functions and the definition of a programmatic interface by which they are accessed. Such an interface would include, but not be limited to functions that supported:

- **Object loading**: the addition of an object file to an ``address space''.
- **Image saving**: the *ld* function of producing an executable is generalized. This would obviate the need for special functions such as ``unexec'' for programs that wish to preserve a running image.

- **Symbol lookup**: obtain the value of a given symbol (perhaps syntactically as a pointer), or in the presence of multiple definitions of a given symbol, the value of any one or all of them.
- **Object "unbinding"**: removal of an object from an address space.
- **Exception handling**: programmatic control of link editor exceptions such as references to undefined symbols.
- **Extensibility**: programmatic interpretation of link editor functions such as relocation.

In addition to their programmatic availability, such functions might also be invoked implicitly by the establishment of conditions through environment variable settings. The functions of interpretation could be enhanced with the (perhaps selective) use of interface descriptions maintained in upgraded object file formats.

These facilities could vastly simplify the use of link-editor properties such as interposition, thereby enabling them to be a more useful tool for the application developer. Consider the "mallometer" example described previously. With a function that performs symbol lookup, the "value-added" version of *malloc* could be written as:

```
char *
malloc(n)
        unsigned int n;
{
        char *cp;
        extern void *ld_lookup();

        ... accumulate statistics about n ...

        /*
         * Look up and call the "next" malloc().
         */
        cp = (char *)(*ld_lookup("malloc", "_next_"))(n);

        ... accumulate statistics about result ...
        return (cp)
}
```

This interposed *malloc* skeleton is an example of use of interposition to invoke the control-flow analog of a UNIX filter. `ld_lookup` is a link editor function that returns a pointer to a (qualified) function name. In this case, the qualifier was `_next_`, a special qualifier meaning the "the *malloc* found by searching dynamically linked objects *after* this one in the symbol precedence order". For the sake of simplicity, disposition of failures has been omitted from the example.

However, since the link editor is interpretive, the reference to the "next" *malloc* might as easily have been coded as:

```
cp = _next_$malloc(n)
```

An unqualified reference to *malloc* would simply invoke the normal symbol lookup rules. Still other qualifiers could be used to identify "absolute" or "fully qualified" function or symbol names. For instance, if a program wanted to call the C language library routine called *malloc*, the reference could be:

```
cp = _c_$malloc(n)
```

Qualifiers need not be exclusively string based of course; these have merely been used in these examples as a suggestion of possible program references.

The interpretive nature of the binding process can also be integrated the activity with other program development facilities. At present, the exception handling facilities in the dynamic link editing process handle errors solely through program termination. Consider a program fragment containing an invocation of *printf* that is erroneously typed in as *pintf.* The execution of such a call would today cause the program to be terminated.[5] An alternative exception-handling facility could be used to intercept such references,

[5] In reality, *ld* anticipates this eventuality and reports it at the time the program is link-edited, thus it is difficult for this scenario

and dispatch them to a debugger, perhaps automatically. The programmer could redirect the erroneous reference to *printf*, and with the assistance of sophisticated software-engineering tools update the source at the same time. In the event that the program referenced some truly undefined (as opposed to misspelled) symbol, an even more sophisticated software engineering environment could support the incremental addition of the missing symbol and the code or data it labeled to the program.

The interpretive nature of the binding process not only permits deferred and possibly interactive bindings, it also provides the opportunity to defer the *implementation* of the binding. For instance, using information either supplied through the environment or programmatically, the dynamic link editor could bind subroutine references not to the final target of the reference, but instead through some intermediate that performed some instrumentation such as ''call graph profiling''.

Alternatively, a function reference could be implemented as something other than a simple subroutine call instruction. This would allow system calls to be defined entirely as library function interfaces. Or, in a general network environment, and if coupled with additional support information such as interface descriptions in object files or configuration databases, the call could be implemented with a Remote Procedure Call (RPC) facility, thereby making the use of distributed resources transparent to the coding of the application and treating it as a problem in application configuration. In this case, the programming environment architecture has treated a symbolic reference as an abstraction that has its semantics defined through execution-time interpretation. In this respect, the symbolic reference is serving the same role for distributed computation as the UNIX file descriptor played in the implementation of transparently networked file systems: a handle for interpreted semantics.

6.3. Future Evolution: Asynchrony

UNIX systems have traditionally provided only one form of asynchrony: the process. Although Berkeley-based systems have introduced a form of asynchronous activity in the support of non-blocking I/O operations and the *select* system call, these are at best crude approximations to truly asynchronous operation. Several applications environments, notably those requiring response to multiple input stimuli (such as window systems or server processes of various kinds) or perhaps requiring real-time constraints, can profit from full support of asynchronous activities.

The traditional approach to providing asynchronous services in a system is to provide specialized interfaces that provide asynchronous operation. For instance, the support of asynchronous I/O operations would be provided by variants on the system calls *read* and *write*. The variants would accept additional arguments describing the disposition of completed I/O requests, and would return almost immediately after being called. The actual transfer would complete some time later.

The implementation of such facilities is generally built to capture the parameters of the requested operation in some form of control block that is used to manage the real I/O activities. On completion, the control block contains a description of how to notify the invoking process that the operation has completed, and what its status was. Other asynchronous interfaces would be similarly constructed, each requiring a specialized control block to capture the required operation and allowing the invoking process to return.

The asynchronous ''control blocks'' created in support of these interfaces can be viewed as the representation of an extremely specialized ''process''. These specialized entities record the state of such processes throughout their ''execution'' (e.g., progress of I/O). Each new form of asynchrony to be added to the system usually involves the creation of a new form of such ''processes''. The specialized nature of their representation often makes each implementation ill-suited to the needs of any later requirement, and over time the system becomes an accretion of such specialized structures.

A view proving popular is that the notion of a ''process'' in UNIX is excessively ''heavy'': a ''process'' is more than just the desired parallel thread of control, it is also an address space, a range of descriptors, and a description of event handling among other things. An alternative notion is that of *lightweight processes* (*lwp*s) [KEPE 85] [TEVA 87], in which the notion of a thread of control is broken out of its UNIX semantics and treated as a distinct entity in itself. A *lwp* can be created efficiently, and multiple *lwp*s can coexist in the address space associated with a single ''heavyweight'' UNIX process.

In an environment supporting *lwp* constructs, it is not necessary to provide specialized interfaces to selectively support asynchronous operation. Any operation can be accessed asynchronously with inexpensively created and managed threads of control. The resulting system is more open to application expansion, as new asynchronous abstractions do not require the services of a ''kernel programmer'' or

to occur in the current system.

clumsy user-program approximations to avoid kernel changes. Note that use of a *lwp*-facility can be "hidden" behind implementations of asynchronous interfaces, and thus any present or future standards requiring the more traditional approach can be easily accommodated.

The programming facilities supported by the *lwp* model of computing provide additional benefits beyond a simple abstraction of asynchrony. *lwps* are "lighter" than traditional UNIX processes because they share heavyweight structures such as address spaces. With such sharing comes mechanisms to manage that sharing, such as *monitors* for the implementation of critical sections, *messages* (often using shared memory) for asynchronous interchange, and *condition variables* for synchronization. In turn, these mechanisms enable the building of reentrant and preemptable code, and render the resulting programming more suitable for execution in a multiprocessor, and more responsive to the preemption demands of real-time and interactive environments.

7. Conclusion

Several changes to the SunOS programming environment have been described. These changes represent a functionally compatible reimplementation of standardized or generally accepted UNIX interfaces. In addition, the changes enrich the environment by providing new capabilities that simplify, and sometimes even render possible, various applications. In still other cases, system and application performance is improved.

The architectural approach of providing simple, fundamental, and general abstractions as primitive mechanisms that are consistently applied has been quite successful. The system has been conceptually simplified while presenting applications programmers with more powerful facilities. In many cases, these programmers have obtained a flexibility previously available only to the "systems programmer". The increased flexibility is demonstrated particularly well with the interpretive binding facilities used to provide shared libraries. The general abstraction of binding mechanisms promises to provide an architectural cornerstone for a wide range of new applications.

These changes show that new implementations of standard functions based on new abstractions are both possible and effective. They also show that the standardizing of clean system *interfaces* is not necessarily a barrier to innovation and, in areas such as dynamic binding, even promotes innovative activity. The resulting and anticipated evolutions have substantially enriched the application programming environment available under SunOS for both ourselves and our user community.

8. Acknowledgements

The ideas and implementations presented here are the product of discussions with and work by many people at Sun. In addition to their contributions to the work itself, Xuong Dang, Jon Kepecs, Joe Moran, and in particular Glenn Skinner provided a great deal of support and assistance through their review of this paper. Other notable participants have included Evan Adams, Bill Shannon, and Richard Tuck.

9. References

[ARNO 86] Arnold, J. Q., "Shared Libraries on UNIX System V", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[DENN 72] Denning, P., S. C. Schwartz, "Properties of the working set model", *Communications of the ACM*, Volume 15, No. 3, March 1972.

[GING 87a] Gingell, R. A., J. P. Moran, W. A. Shannon, "Virtual Memory Architecture in SunOS", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[GING 87b] Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, "Shared Libraries in SunOS", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[JOY 83] Joy, W. N., R. S. Fabry, S. J. Leffler, M. K. McKusick, *4.2BSD System Manual*, Computer Systems Research Group, Computer Science Division, University of California, Berkeley, 1983.

[KEPE 85] Kepecs, J. H., "Lightweight Processes for UNIX: Implementation and Applications", *Summer Conference Proceedings, Portland 1985*, USENIX Association, 1985.

[KLEI 86] Kleiman, S. R., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[MORA 88] Moran, J. P., "SunOS Virtual Memory Implementation", *Spring Conference Proceedings, London 1988*, European UNIX Users Group, 1988.

[MURP 72] Murphy, D. L., ''Storage organization and management in TENEX'', *Proceedings of the Fall Joint Computer Conference*, AFIPS, 1972.

[ORGA 72] Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, 1972.

[RITC 74] Ritchie, D. M., K. Thompson, ''The UNIX Time-Sharing System'', *Communications of the ACM*, Volume 17, No. 7, July 1974.

[STRO 86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley Publishing Company, 1986.

[TEVA 87] Tevanian, A., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, M. W. Young, ''Mach Threads and the UNIX Kernel: The Battle for Control'', *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

POSIX - A Standard Interface

Jim R Oldroyd

The Instruction Set
jr@inset.co.uk

ABSTRACT

There are a bewilderingly large number of UNIX systems in existence today. Most are derived from one of two main ''flavours'' of the operating system - System V and 4.*BSD.

However, these derivatives vary considerably in a variety of ways, both expected and unexpected. Differences exist in the behaviour of functions, their types, the type and number of arguments, location and contents of header files, etc; also the commands and utilities may differ or take different options etc.

These differences provide headaches to authors of *portable* applications. Although it is possible to write software that will compile without modification and run correctly on a large number of existing systems, considerable expertise and knowledge of the different systems is required to do this. Acquisition of this expertise can be a time-consuming and costly overhead.

Developing software which is portable across different UNIX operating systems suffers from a major problem: the software is still likely to need modification when another new implementation of the system appears.

The POSIX System has been developed to ease this problem. The interfaces (system calls, libraries and commands) described in POSIX have evolved from those on existing UNIX systems and, where things differ on existing systems, the POSIX interfaces represent a compromise or an improvement.

The POSIX interface can be implemented on all existing UNIX systems (and, in fact on non-UNIX systems too). Porting applications to new systems will be considerably simplified, if both source and target systems are POSIX compatible.

This paper presents a technical overview of POSIX and looks at areas which differ from existing systems. The paper takes the view of an Applications Writer, but in so doing, highlights areas which will be of interest to those responsible for making a system *POSIX compatible* .

An overview of the position of POSIX and impact in the market place is also given.

1. What is POSIX ?

1.1. History

Since the creation of the UNIX system by Ken Thompson and Dennis Ritchie in 1969, many different versions of the system have evolved. Each version provides facilities which improve on previous versions, or which address specific needs which competing versions do not address.

At the start of 1985, hundreds of varieties of UNIX systems existed, supplied by a large number of manufacturers. Some of these systems replaced older versions, and added new features. Some systems from different suppliers looked similar, others did not. For one machine, at least twenty different commercially available versions of the operating system were available (and still are).

This proliferation of variants on a theme means that a potential user of the system is free to choose exactly that system which is best able to do the job required of it at the most suitable cost.

Unfortunately, however, during the years over which the development of all these systems has taken place, many different institutions and many different individuals have influenced the flow of ideas. This has naturally lead to systems having obvious differences, even in those areas which are superficially common.

This leads to a very real difficulty: non-portability of applications between systems.

An applications developer would have to spend quite some time converting the application so that it runs on another version of the system. Now, as anyone who's ever done this knows, porting and re-porting is time consuming, it is boring, can be very tricky, and consequently it is expensive. In fact, in some cases, the cost of porting an application to another version of the system can be so expensive, that, even today, many applications are only available on certain versions of the system.

Now, this is clearly not the best commercial position for application developers. So, in 1984 a group of system suppliers joined together to form the X/Open Group[1]. The intention of the group was to specify a common version of the system so that applications written to that *common denominator* would be portable in source code form across all the member's systems (and any others which were compatible). This common definition was nothing new – it was a specification of the common parts of all the existing systems. The definition was published in the X/Open Green Book[2], and now forms the basis of the Common Applications Environment which covers programming languages, data management, internationalisation, terminal control and source code transfer as well as the operating system interface definition.

1.2. IEEE P1003

In 1985, the Institute of Electrical and Electronics Engineers created the P1003 committee with the task of specifying a Portable Operating System Interface for Computer Environments. The P1003.1 sub-committee became responsible for the specification of the base operating system, while other sub-committees had responsibilities for the command interface, verification, real-time and security aspects, see **Other P1003 Activities.**

At the time of writing, the P1003.1 committee has produced a Trial-Use Standard[3] and now has a document (Draft 12) which is undergoing Full-Use Ballot. On acceptance, the document will be published, and becomes an IEEE/ANSI Standard. The Operating System described in the document (POSIX) is then likely to become *the* standard interface for computer systems. Many vendors have contributed to the development of POSIX System, with the result that it contains much that can be found in existing products. It also contains a number of improvements in areas where existing systems were known to have problems, and it contains some simplifications.

There is considerable commitment to the Standard. Many vendors have announced intentions to provide POSIX compatible systems. The X/Open Group has announced that the relevant parts of it's Common Applications Environment will be made to comply. Some companies are known to have been tracking the draft standard in their product development cycles.

So what exactly is the POSIX System?

It is based on the 1984 */usr/group* standard[4] which evolved from v7[5] and 4.2 BSD[6]. The P1003.1 committee further developed the 1984 Standard. As it evolved, the behaviour of many suppliers' existing systems was taken into account. The specification was adapted so that, where possible, as many existing products as possible would be compatible[7].

The result is a system which will look very familiar to experts in traditional UNIX systems. The remainder of this paper takes a look at the various concepts and interfaces described in the standard.

---

1 At formation, the group consisted of Bull, ICL, Nixdorf, Olivetti and Siemens. The group has now been expanded by the membership of AT&T, DEC, Ericsson, H-P, NCR, Philips, Sun Microsystems and Unisys.

2 Actually, the X/Open Portability Guide.

3 Draft American National Standard, April 1986, ISBN 0-471-85027-6.

4 /usr/group in the US has many active technical groups which produce various documents. Their UNIX standards work stimulated the formation of the IEEE P1003 group.

5 Developed at Bell Laboratories, the first UNIX to emerge outside was version 5, or v5. Internal Bell versions still use the original version scheme; the current version is v9.

6 BSD UNIX is available direct from the University of California, Berkeley and also from many suppliers who have incorporated it into their products.

7 Many of these were derived from AT&T's System V product, documented in the System V Interface Definition (SVID). The SVID referred to throughout this paper is Issue 2, published in 1986, comprising three volumes.

2. Overview of the Standard

The Standard consists of 10 chapters and 3 appendices.

Chapters 1 and 2 introduce the standard, discuss conformance issues and define global concepts, error codes and constants.

Chapters 3 to 7 contain descriptions of the system interfaces. They are arranged by functional category, rather than in alphabetical order – the index is an essential tool to assist in locating something!

Chapter 8 describes the bindings of the system to the C language. Actually, the bindings are described throughout the standard, making POSIX heavily dependent on the C Language. Bindings to other languages, and a language-independent description are being developed, and will be published later.

Chapter 9 describes the interfaces to system databases. For security reasons, only interfaces are described – the location and full contents of such databases is not mentioned.

Chapter 10 discusses data interchange. The format in which data should be written on a media for transport to foreign systems is described.

Appendix A mentions other Standards, and the relationship of this one to others. Lists of addresses of relevant bodies are given.

Appendix B is the Rationale. This lengthy disquisition lists many of the reasons why things are the way they are, and also discusses why many ideas were not included. In some areas it is sketchy, but in others it is very useful.

Appendix C is a comparison with the SVID. For a number of years, the SVID has been used in the commercial world as the definition for portable systems. This appendix lists areas where POSIX has deviated from the SVID.

3. Scope and Conformance

The POSIX Standard describes the interfaces to the operating system which are traditionally known as *system calls*. In the UNIX world, these have always been published as section 2 of the Programmer's Reference Manual.

In addition, the standard requires that (for the C language, at least) a large number of additional functions are provided. Once ANSI publishes its C language definition[8], it will become a requirement that an ANSI conformant compiler is used for POSIX conformant C language applications. The ANSI definition of the C language defines functions such as *printf()*, *getchar()*, *strcpy()*, etc to be part of the C language. Until such compilers are available, all C compilers used for POSIX applications must provide all the functions which the ANSI definition requires. This amounts to almost all the functions in the traditional standard C libraries, plus a few others. These are described later, see **C Language Functions**.

A C language application which uses only the above interfaces (and which does not mis-use the C language) will be known as a *Strictly Conforming Application*. This means it should be able to compile and run (with no modifications at all) on all POSIX conformant systems.

A C language application which uses the above interfaces and, additionally, those described in any other ANSI standard (e.g., other programming languages, graphics, etc) will be known as a *Conforming Application*. Naturally, for it to compile and run, the system will have to conform to all the required standards.

An application which uses the above interfaces and, additionally, other interfaces which are not described in any standard, will be known as a *Conforming Application Requiring Extensions*. It will have to list all extensions it needs, so that they may be made available on target systems.

As far as a system is concerned, in order to provide a POSIX *conformant implementation*, all the system calls and library interfaces documented must be provided. Some interfaces are *optional*; the system may provide such interfaces if desired, but they are not obliged to be provided.

There is no requirement for the system to be written in any particular programming language.

8 ANSI X3.159-198x, currently undergoing second public ballot.

4. Technical Contents of POSIX

The interfaces to the operating system are described in chapters 3 to 7. Additional interfaces are described in chapters 8 and 9. The Data Interchange Format is described in chapter 10.

Chapter	*Contents*
2	Definitions and General Concepts
3	Process Primitives
4	Process Environment
5	Files and Directories
6	Input / Output
7	Terminal Handling
8	C language functions
9	System Databases
10	Data Interchange Formats

Table 1. Technical Chapters in POSIX

Below, the contents of each of these chapters is explained, and some guidelines as to any differences between the POSIX interfaces and those on existing BSD and SVID systems are given.

It is important to note that, the POSIX being described here is that contained in the Draft 12 document, currently undergoing ballot. As much up-to-date information as possible has been included at the time of writing (January 1988). It is possible, and indeed quite likely, that the balloting committee may make alterations to the definition; the extent of such alterations cannot be predicted.

5. General Concepts

5.1. Errno Values

POSIX specifies *errno* values for all of the interfaces described. These are values returned in the external variable *errno* in the event of failure of a request to the system. Each value indicates a different reason for failure.

Extreme care was taken during the early stages of design of the standard to ensure that the values returned were consistent with those values in use on traditional implementations. In some cases conflicts arose, in which case the ''correct'' solution was chosen.

The standard does not give the actual numeric values to be used for *errno*. Rather, it *names* the values using symbolic constants, and leaves it to each implementation to define values. This ensures source code portability of applications which use the symbolic constants.

Implementors of POSIX systems should check all interfaces, as some changes to the *errno* values may be needed to most existing systems.

5.2. Limits

There have always been a number of items for which limits exist, such as the maximum number of files a process may have open simultaneously, or the maximum number of characters in a file or path name, etc.

POSIX recognises this fact, and defines a set of limits by means of symbolic constants. That is to say, the standard will indicate, for example, that a failure may occur if the action would result in the number of files open exceeding `{OPEN_MAX}`.

Most of the ''traditional'' limits exist. Some (the system-wide limits such as `{LOCK_MAX}`) were omitted, as they are of no use to an individual process – it may not be able to have up to `{LOCK_MAX}` locks due to some being in use by other processes. Some new limits exist; these were adapted from the ANSI C language standard and define the minimum and maximum values for language types which were previously undefined (e.g., `{UCHAR_MAX}`, `{UINT_MAX}`, etc).

The actual values for these constants are not given, but the standard does list minimum acceptable values. I.e., all implementations must support at least the minimum value given for each limit.

It is recognised, however, that on real systems, it is usual for larger systems in the range to provide greater values for some limits, e.g., `{PID_MAX}` or `{OPEN_MAX}`, etc. For applications, binary-ported to a larger model of a range, to be able to determine the actual value of a system limit is very useful. To this end, POSIX defines a new function, *sysconf()* which allows this information to be obtained, see **Process**

Environment.

Furthermore, some limits are file system dependent, and may vary on one system. For example, `{NAME_MAX}` (the maximum length of a file name) is 14 on SVID file systems and 255 on BSD file systems. On a system which allows both types of file system to be networked together in an NFS[9] environment, the value of `{NAME_MAX}` may vary along a path. As this could disadvantage applications which may wish to optimise their use of system resources, POSIX defines a second new function, *pathconf()* which returns the value of a "limit" for a given path. A further interface, *fpathconf()* exists to allow the information to be obtained for an open file descriptor. See **Files and Directories**.

5.3. Headers

POSIX requires the existence of several headers, used to declare data types, types of functions and define structures used by the functions.

These exist as files on most traditional systems, and the POSIX versions do not differ greatly from existing systems. Nevertheless, some changes will be necessary for POSIX conformance: the location of the file may need changing, some additional constants may be needed, some additional data types may have to be added and some structures may need slight modification in line with the amended data types.

5.4. Alternate Behaviours

There are several areas in which traditional systems behave differently. One example is the question of with which group a file is created. On SVID systems, it is the effective group ID of the process which creates the file; on BSD systems it is the group ID of the directory in which the file is being created.

In this and other areas, the POSIX system allows either form of behaviour of a system. Applications wishing to be POSIX conformant must be able to tolerate both forms of behaviour. For some alternates, an application can predict the behaviour of a system by using a "flag" defined in **<unistd.h>** or available from *sysconf()* or *pathconf()*. In other cases, a flag is not provided – in such cases, the application should assume either style of behaviour, and take steps to ensure that the final result is what is desired.

6. Process Primitives

Chapter 3 describes the following interfaces. Table 2 lists their names, shows the status in POSIX, and lists whether differences exist between the POSIX interface and that of BSD and SVID.

9 NFS™ or Network File System, developed by Sun Microsystems, allows file systems on other systems to be mounted into the local file system tree. To applications, they look almost exactly like local file systems.

Interface	*Status*	*Changes to BSD*	*Changes to SVID*
fork ()	M[11]	–	–
execl ()	M	–	–
execv ()	M	–	–
execle ()	M	–	–
execve ()	M	–	–
execlp ()	M	–	–
execvp ()	M	–	–
wait ()	M	–	–
wait2 ()	M	new	new
_exit ()	M		–
kill ()	M	yes	–
siginitset ()	M	yes	new
sigfillset ()	M	yes	new
sigaddset ()	M	yes	new
sigdelset ()	M	yes	new
sigismember ()	M	yes	new
sigaction ()	M	yes	yes
sigprocmask ()	M	yes	yes
sigpending ()	M	new	new
sigsuspend ()	M	yes	new
alarm ()	M	–	–
pause ()	M	–	–
sleep ()	M	yes	–

Table 2. Process Primitives in POSIX order

The *wait2()* interface is new in POSIX. It is similar to the *wait3()* interface of 4.3 BSD, but *wait2()* lacks the third argument, the returned resource usage summary. On BSD, it is thus simple to implement *wait2()* as a library routine. On SVID systems, implementation is more complex. The routine provides the ability to return immediately, if no children have exited. SVID systems do not currently provide the ability to do this, so this will require kernel changes.

From the table it is clear that the signal handling interfaces are not the same as on existing systems. The traditional *signal()* interface has been replaced with a set of interfaces which implement ''reliable'' signals. The interfaces are derived from those on 4.3 BSD, but are not the same. Those interfaces were, in turn, derived from the 4.1 BSD interfaces which are also in System V.3. ''Reliable'' signals give the application the ability to block signals (i.e., prevent their delivery, without losing them), to have signals automatically blocked on entry to a signal handler, and to manipulate sets of signals together. Unfortunately, POSIX does not require that blocked signals are queued, i.e., if a second signal of any particular type occurs, only one need be delivered to the application. However, queuing is not prevented by POSIX. This issue affects Real-Time, and is being addressed by the Real-Time Group, P1003.4 (see **Other P1003 Activities**).

There are some modifications to be made to 4.3 BSD systems to implement these signal routines. More extensive changes will be needed on 4.1 and 4.2 BSD and System V.3. Considerable work will be needed on System V.2 and other systems which do not have any code to do blocking and grouping of signals.

Kill() in POSIX has similar semantics to the SVID kill. Specifically, for a process to have permission to send a signal to another, the receiving process's real or effective user ID must match the real or effective user ID of the sending process. This allows a user to send a signal to a setuid process. On BSD, both sending and receiving processes must have the same *effective* user ID. Some work will therefore be needed to adapt BSD systems.

Sleep() has a return value, not in BSD. The value is the amount of unslept time in the event that the sleeper was aroused prematurely by an incoming signal.

11 In this and subsequent similar tables, M means mandatory and O means optional.

7. Process Environment

Chapter 4 describes the following interfaces:

Interface	*Status*	*Changes to BSD*	*Changes to SVID*
getpid ()	M	–	–
getppid ()	M	–	–
getuid ()	M	yes	yes
geteuid ()	M	yes	yes
getgid ()	M	yes	yes
getegid ()	M	yes	yes
setuid ()	M	yes	yes
setgid ()	M	yes	yes
getgroups ()	O	–	new
cuserid ()	M	new	–
getpgrp ()	M	yes	–
setpgrp ()	M	yes	–
jcsetpgrp ()	O	new	new
uname ()	M	new	–
time ()	M	yes	yes
times ()	M	yes	yes
getenv ()	M	–	–
ctermid ()	M	–	–
ttyname ()	M	–	–
isatty ()	M	–	–
sysconf ()	M	new	new

Table 3. Process Environment Interfaces in POSIX order

There seem to be quite some changes to what are fairly simple interfaces in this section. On closer examination, the changes are not significant. For all the *yes*es above, except those noted below, the "change" is simply that the type of the argument or return value is now a defined type. The types *uid_t*, *time_t* and *clock_t* have been defined. It should suffice to add them to **<sys/types.h>**.

The SVID interfaces *cuserid()* and *uname()* need to be added to BSD systems.

Getpgrp() and *setpgrp()* differ from those the BSD versions in that in POSIX it is not possible to obtain or change the process group of another process. This change can be implemented as a library routine.

The *sysconf()* routine is new in POSIX It provides the application with a means of obtaining the value of a system configuration limit or behavioural characteristic. For example, the actual value of the "constant" `CHILD_MAX` or `UID_MAX`, etc, or the value of the "flags" `_POSIX_JOB_CONTROL` or `_POSIX_EXIT_SIGHUP`, etc. The values of the system configuration limits must be equal to or greater than set minimum values given in Chapter 2; but applications frequently may wish to make use of additional resources of some systems, rather than to restrict themselves to the known minima.

Implementation can be done in several ways. On systems (or ranges of systems) on which these values are fixed, *sysconf()* could be a library routine which prints out the fixed values from **<limits.h>** and **<unistd.h>**. On systems where the values may vary, the system configuration and/or bootstrap procedures could create a table of values in a file which is used by the *sysconf()* library routine. Alternatively, *sysconf()* could be a system call which returns the desired information from the kernel.

Two major areas are worth some extra notes:

7.1. Job Control

Job Control is an option in POSIX. That is, systems do not have to support it, but if they do, they shall support it as described.

The job control described "looks" similar to that of BSD in that processes can be suspended, and later restarted in the foreground or background. There exists the ability to have background processes suspended on attempting to write to the controlling terminal. These semantics differ from those of the SVID's layered shell, and are not supported on existing SVID systems.

The interfaces described in POSIX differ in some respects to those of BSD. The reason for the changes was to simplify implementation on SVID systems. Unfortunately, a recent investigation of the POSIX job control and process groups by BSD implementors has shown that the POSIX descriptions are very vague about some aspects: how many process groups exist, what the precise characteristics of a process group are, and what the exact behaviour of some interfaces should be.

This discovery has lead to considerable debate amongst the POSIX committee and interested parties concerning the whole area of job control, process groups, sessions and terminals. At this time, the debate continues. It can be expected that some (major?) clarifications will occur to Draft 12 in these areas before publication. The exact nature of the interfaces can then be investigated.

SVID systems do not support this job control option.

7.2. Multiple Group Support

The ability for a process to be a member of several groups at once (*à la* BSD) exists, as an option, in POSIX.

However, this too, is subject to clarification before publication of the standard, although the semantics of multiple groups are not expected to differ much from BSD.

SVID systems do not support multiple groups.

8. Files and Directories

Chapter 5 describes the following interfaces:

Interface	*Status*	*Changes to BSD*	*Changes to SVID*
opendir ()	M	–	–
readdir ()	M	yes	–
rewinddir ()	M	–	–
closedir ()	M	yes	yes
chdir ()	M	–	–
getcwd ()	M	–	–
open ()	M	yes	yes
creat ()	M	–	–
umask ()	M	–	–
link ()	M	–	–
mkdir ()	M	–	new
mkfifo ()	M	new	new
unlink ()	M	–	–
rmdir ()	M	–	new
rename ()	M	–	new
stat ()	M	–	–
fstat ()	M	–	–
access ()	M	–	–
chmod ()	M	–	–
chown ()	M	–	–
utime ()	M	–	–
pathconf ()	M	new	new
fpathconf ()	M	new	new

Table 4. File and Directory Interfaces in POSIX order

The directory access routines (*directory*(3)) are included in POSIX. They are derived from the SVID version. System V.2 does not provide them, but they can easily be added as library routines. On BSD, they are provided, but the name of the header, and data structure used are different. However, the POSIX version can be provided with no loss of compatibility.

Open() differs from all existing systems in that it has a new flag, O_NONBLOCK, to provide non-blocking I/O. This differs from the SVID and BSD's O_NDELAY flag (which are both different) as follows. On SVID systems, O_NDELAY does not distinguish between the end-of-file and no-data-available conditions (*read()* will return zero in both cases). This is a problem for applications wishing to use asynchronous I/O techniques. The BSD version does differentiate these two conditions, in that on no-data-available, a *read()*

returns −1, and sets *errno* to `[EWOULDBLOCK]`. The POSIX `O_NONBLOCK` takes the BSD functionality as it is more useful to applications. The error value is changed to `[EAGAIN]` as the BSD value is arguably ''wrong''. SVID systems will therefore need some work; BSD systems need an *errno* change.

The BSD interfaces *mkdir()* and *rmdir()* were incorporated into the SVID for System V.3. They will need adding to System V.2 systems. *Rename()* will have to be added to all systems; it comes from the ANSI C language definition. These can all be implemented as library routines which make use of existing system calls.

The interfaces *pathconf()* and *fpathconf()* are new in POSIX. Similar to *sysconf()*, see **Process Environment**, these interfaces allow an application to determine the actual value of a path dependent limit. *Pathconf()* takes a pathname argument, whereas *fpathconf()* uses an open file descriptor. An example would be *pathconf(path, _PC_NAME_MAX)* which may return 14 if *path* were */usr/grp*, but may return 255 if *path* were */usr/grp/jr*. This could happen if the latter directory were an NFS mount point. Another example is *fpathconf(0, _PC_MAX_INPUT)* which returns the maximum input line length for file descriptor 0. This could vary depending on the configuration of a terminal driver, or whether file descriptor 0 references a file or pipe.

8.1. Fifo Files

The concept of a fifo (first-in-first-out file) comes from the SVID. A fifo looks exactly like a pipe, in that one process writes to it, and another process reads from it; the data being received in the order it was written. One all data has been read, the file is empty again. The difference between a fifo and a pipe is that a fifo has a name and looks like a file on the file system. This facility allows anonymous rendezvous of processes – very useful for some applications.

The interface to these files is *mkfifo()*. This does not exist on any system, but is implementable as a trivial library routine on SVID systems. On 4.3 BSD it can probably also be implemented in user space by making use of UNIX domain sockets. This may not work on 4.2 BSD, due to bugs in the socket code.

9. Input / Output

Chapter 6 describes the following interfaces:

Interface	*Status*	*Changes to BSD*	*Changes to SVID*
pipe ()	M	–	–
dup ()	M	–	–
dup2 ()	M	–	yes
close ()	M	–	–
read ()	M	yes	yes
write ()	M	yes	yes
fcntl ()	M	yes	–
lseek ()	M	–	–

Table 5. I/O Interfaces in POSIX order

The important change here is to the *read()* and *write()* interfaces. BSD systems need simply to add the new `O_NONBLOCK` flag name and change the *errno* value returned in the no-data-available situation. SVID systems will have to change so that an error (rather than 0) is returned in the no-data-available situation.

The *dup2()* interface is in the SVID, and is compatible. However, it does not exist in System V.2 systems. On these systems, it is implementable as a library routine which calls *fcntl()*.

Fcntl() is included in POSIX for both file control and advisory locking. Earlier versions of POSIX specified *lockf()* for locking, but this was dropped as it suffers from some deficiencies. In particular, *lockf()* does not allow multiple read locks on a segment – all locks are exclusive. Furthermore, locks on files only open in read mode (`O_RDONLY`) are not supported. As this is non-optimal from the point of view of applications, and as the SVID locking provided in *fcntl()* already supports the above, it was adopted instead. Some work will be needed on BSD systems to support this.

10. Terminal Handling

Chapter 7 describes the following interfaces:

Interface	*Status*	*Changes to BSD*	*Changes to SVID*
cfgetospeed ()	M	new	new
cfsetospeed ()	M	new	new
cfgetispeed ()	M	new	new
cfsetispeed ()	M	new	new
tcgetattr ()	M	new	new
tcsetattr ()	M	new	new
tcsendbreak ()	M	new	new
tcdrain ()	M	new	new
tcflush ()	M	new	new
tcflow ()	M	new	new
tcgetpgrp ()	O	new	new
tcsetpgrp ()	O	new	new

Table 6. Terminal Control Interfaces in POSIX order

As first glance, this may seem completely new. It is! Application writers may now be suffering from some degree of shock at the prospect of recoding all their programs. System implementors may well be somewhat worried too. But, for implementors, or at least, those from SVID environments, things are not too bad.

POSIX provides terminal control through a functional interface and a *termios* data structure.

10.1. Termios

The *termios* structure is very similar to the *termio* structure of the SVID. All the *termio* features are supported. The only differences are that certain control characters no longer occupy the same space (which didn't matter, as they were mutually exclusive anyway), and that the additional constants B19200 and B38400 are defined for baud rates.

It is believed that *termios* can be implemented on SVID systems as a library interface. This is no consolation for BSD systems, where a new terminal line discipline will have to be provided.

Termios does allow for split speed I/O (different input and output baud rates). However, support of split speeds is not mandatory (although it is useful).

The application interface to *termios* is through the functional interfaces listed above. The traditional *ioctl()* interface is not described in POSIX. Applications should use the first four functions to get and set baud rates and *tcgetattr()* and *tcsetattr()* to get and set all other attributes (rather like *ioctl(),* actually).

Tcsendbreak() sends a ''break'' for a specified period – this functionality seems to be mandatory. *Tcdrain()* suspends the process until all output written has been transmitted and *tcflush()* discards unread or untransmitted data. *Tcflow()* is used to implement XON/XOFF flow control.

The two *tcXetpgrp()* functions are optional, but are needed to support job control. Their purpose is to set the distinguished process group (i.e., foreground process group). As much concerning job control is still being clarified (see **Job Control**), the descriptions of these functions may change prior to publication of the standard.

11. C language Functions

Chapter 8 describes the interface to the C programming language.

When the ANSI X3.159-198x Programming Language C Standard is published, it will be the basis for a C language for POSIX conformant C language applications. As the publication of X3.159 seems far removed at this time, POSIX allows for applications to be written using existing C compilers which support the standard C and math libraries. It is recognised that the lack of an adopted C language standard negatively affects the ability of applications developers to write portable applications, but it is suggested that the most recent draft of the proposed ANSI standard is used as a guideline to maximise future portability.

The interfaces which must be supported are:

abort ()	*floor* ()	*isupper* ()	*sin* ()
abs ()	*fmod* ()	*isxdigit* ()	*sinh* ()
acos ()	*fopen* ()	*ldexp* ()	*sprintf* ()
asctime () ♦	*fprintf* ()	*localtime* () ♦	*sqrt* ()
asin ()	*fputc* ()	*log* ()	*srand* ()
assert ()	*fputs* ()	*log10* ()	*sscanf* ()
atan ()	*fread* ()	*longjmp* ()	*strcat* ()
atan2 ()	*free* ()	*malloc* ()	*strchr* ()
atof ()	*freopen* ()	*modf* ()	*strcmp* ()
atoi ()	*frexp* ()	*perror* ()	*strcpy* ()
atol ()	*fscanf* ()	*pow* ()	*strcspn* ()
bsearch ()	*fseek* ()	*printf* ()	*strftime* ()●
calloc ()	*ftell* ()	*putc* ()	*strlen* ()
ceil ()	*fwrite* ()	*putchar* ()	*strncat* ()
clearerr ()	*getc* ()	*puts* ()	*strncpy* ()
cos ()	*getchar* ()	*qsort* ()	*strpbrk* ()
cosh ()	*getenv* () ♦	*rand* ()	*strrchr* ()
ctime () ♦	*gets* ()	*realloc* ()	*strspn* ()
exit ()	*gmtime* () ♦	*remove* ()●	*strstr* ()
exp ()	*isalnum* ()	*rename* () ♦	*strtok* ()
fabs ()	*isalpha* ()	*rewind* ()	*tan* ()
fclose ()	*iscntrl* ()	*scanf* ()	*tanh* ()
fdopen ()	*isdigit* ()	*setbuf* ()	*time* () ♦
feof ()	*isgraph* ()	*setjmp* ()	*tmpfile* ()
ferror ()	*islower* ()	*setlocale* () ♦●	*tmpnam* ()
fflush ()	*isprint* ()	*siglongjmp* ()●	*tolower* ()
fgets ()	*ispunct* ()	*signal* ()	*toupper* ()
fileno ()	*isspace* ()	*sigsetjmp* ()●	*ungetc* ()

Table 7. Required Standard C Library Interfaces

Some of the above interfaces are documented fully both in POSIX and in the ANSI C language document. This is because there are further specifications or amplifications in POSIX over the ANSI version. These routines are marked with a ♦ symbol. Others do not exist on traditional systems (marked ●), but are in the ANSI definition. The remaining interfaces are listed in POSIX, but left undefined; however, they should conform to traditional definitions or be in line with ANSI. Of the remaining interfaces, many have been more rigorously specified in ANSI than on traditional systems, and so may need improving when an ANSI C compiler is adopted.

11.1. Internationalisation

The POSIX system has been developed at a time when there is much international interest concerning support for applications which have to run correctly across international boundaries. In recent years, various groups[12] have been active in developing interfaces for use by internationalised applications.

Most of the impact of internationalisation does not concern the POSIX definition. Two interfaces are documented, however.

The first is *setlocale()*. This is used to specify a native environment for the application. The locale used is under user control by means of environment variables, which an application should read and pass to *setlocale()*. Once the locale is set, several other functions will perform in a locale dependent manner. The extent to which functions pay attention to their locale is not defined by POSIX, but it will be defined by other groups and manufacturers. Furthermore, POSIX does not specify which locales are supported; only that one "standard" locale is supported on all systems[13].

12 /usr/group and X/Open, to name but two.

13 Actually, it is the P1003.2 Commands and Utilities standard which specifies that at least the "C" locale is supported. This locale defines a minimum character set and a collating sequence; P1003.2 may well additionally specify one language.

Secondly, POSIX specifies support (via *strftime()*) for time zone dependencies. As existing time zone notification mechanisms have proven inadequate, POSIX requires that systems accept a TZ environment variable containing either of the following forms of data:

> TZ=*std offset* [*dst* [*offset*][,*start* [/*time*],*end* [/*time*]]]
> TZ=/*string*

The first form provides a very flexible means of specifying the names of the time zone, offset from GMT, start and end of daylight saving time, and offset thereof. Start and end times can be specified in various forms, including "Saturday at 02:00 on the third week of April". The second form allows for local variations.

12. System Databases

The traditional password and group files have been replaced with the User and Group Databases in POSIX, following input from the Security working party.

The name of the databases are not given, however, applications may access them using the following routines, described in Chapter 9:

Interface	*Status*	*Changes to BSD*	*Changes to SVID*
getgrent ()	M	–	–
getgrgid ()	M	–	–
getgrnam ()	M	–	–
setgrent ()	M	–	–
endgrent ()	M	–	–
getpwent ()	M	–	–
getpwuid ()	M	–	–
getpwnam ()	M	–	–
setpwent ()	M	–	–
endpwent ()	M	–	–

Table 8. System Database Access Interfaces in POSIX order

These interfaces behave as on traditional systems. The only difference is that for security reasons, the encrypted password field of traditional systems is not described in POSIX.

13. Data Interchange

The intention of Chapter 10 is to document a Data Interchange format which can be used when writing files to a transportable media for input on another POSIX system.

The standard mentions that systems should provide as many as possible actual hardware media (e.g., tape, floppy etc) and lists hardware formats, but it is outside the scope of POSIX to make support of any type of hardware mandatory.

Chapter 10 includes both the *cpio* and an improved *tar* format, known as *ustar* (Uniform Standard Tar). The intention was that eventually, one of the two formats would be selected as the only required format. Unfortunately, there were active proponents for each format, and it has not yet been possible to select just one. It now seems likely that both *cpio* and *ustar* will be required to be supported!

14. Publication of the Standard

Currently POSIX P1003.1 is undergoing Full-Use ballot. The ballot is now closed (mid-January 1988), and resolution of objections is in progress. It is anticipated that a list of proposed changes to Draft 12 will be recirculated to the ballot committee for approval in February.

If they are approved, the changes could be applied to the document and the document submitted to the IEEE Standards Board for ratification in March or April. Publication as a standard could then occur by June or July, 1988.

Due to the complex verification and administrative procedures employed by standards people, it is possible that this prediction is wildly inaccurate.

In parallel with the current Full-Use ballot, the draft standard has been submitted to the International Standards Organisation (ISO) for approval as a full International Standard (IS). This approval is expected to be given, and the document could become an IS within two years.

15. Other P1003 Activities

The P1003 group currently consists of seven sub-committees:

Committee	*Activity*
P1003.1	POSIX, System Calls and Libraries Standard
P1003.2	POSIX, Commands and Utilities Standard
P1003.3	Verification
P1003.4	Real-Time Extensions to POSIX
P1003.5	ADA Bindings to POSIX
P1003.6	Security Extensions to POSIX
P1003.0	How to use P1003.1 – P1003.6 together

Table 9. P1003 Sub-Committees

Only the P1003.1 activity is nearing completion. The position of P1003.2 is described below.

P1003.3 is producing a document outlining how to test conformance to the POSIX standard. Their work is tracking P1003.1 and P1003.2.

The P1003.4 and P1003.6 committees working on Real-Time and Security Extensions are expected to produce documents at some time in the future. It is anticipated that these documents will be of the form ''For a Real-Time POSIX system, implement P1003.1 and apply the following changes and additions...''.

The P1003.5 committee is working on ADA bindings. Their work covers two areas, specifying the interfaces to POSIX for ADA language applications, and secondly, they are attempting to determine if there could be any problems in implementing a POSIX system in ADA.

The P1003.0 group is expected to produce a short overview document explaining the relationship between the other documents and how to use them together.

16. The P1003.2 Commands and Utilities Standard

P1003.2 is specifying a standard interface to commands and utilities. They are working along these lines:

1. A list of utilities to be included has been selected. The basis for inclusion was that the utility is needed for shell scripts, or for installing source code.
2. A description of each utility is being produced. The description contains much more detail of the behaviour of the utility than traditional documentation. For example, the exact output strings of each utility are being documented to allow post-processing. (Error messages are not documented.)
3. Options which differ across existing systems, or which are of little use, are being dropped. Only in circumstances where this would remove important functionality, are they being included.

The work is currently about one third complete. Publication of P1003.2 should not be expected until the end of 1989.

17. Impact on the Market

Many organisations have already expressed support for POSIX. Many systems suppliers are known to be working on POSIX P1003.1 compliant systems. It should be expected that some suppliers announce products very soon after POSIX is published.

One added impetus is the announcement of X/Open (in March 1987), the American National Bureau of Standards (NBS) (in September 1987) and the British Central Computer and Telecommunications Agency (CCTA) (in January 1988) that they support POSIX. The NBS specifies purchase guidelines to all Federal Government Agencies wishing to purchase anything – what they purchase must conform to the NBS guidelines. As the US Government is the largest customer in the world, the fact that the NBS has endorsed POSIX is of great importance to many system manufacturers and software suppliers.

Many hundreds of organisations have contributed toward the development of POSIX. It can be expected that future releases of UNIX based and other systems from most suppliers will be POSIX compatible, or support a POSIX environment. Systems may offer additional features which do not confilict with POSIX – that is what will make them attractive to the customer. Such extension would be usable for the specific purposes which they serve, but obviously, an application requiring maximum portability could not use them.

18. References

1. POSIX P1003.1. The Institute of Electrical and Electronic Engineers Inc, 345 East 47th Street, New York, NY 10017, USA. Draft 12, October 12, 1987.
2. POSIX P1003.2. The Institute of Electrical and Electronic Engineers Inc, 345 East 47th Street, New York, NY 10017, USA. Draft 4, November 16, 1987.
3. ANSI X3.159-198x, Programming Language C. American National Standards Institute, CBEMA, 311 First Street NW, Suite 500, Washington DC 20001, USA.
4. X/Open Portability Guide. Elsevier Science Publishers BV, PO Box 1991, 1000 BZ Amsterdam, The Netherlands. Issue 2, January 1987, ISBN 0-444-70179-6.
5. 1984 /usr/group Standard. /usr/group, 4655 Old Ironsides Drive #200, Santa Clara, CA 95054, USA.
6. UNIX Programmers Reference Manual (PRM), 4.3 Berkeley Software Distribution. Computer Science Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, USA. 4.3 version, April 1986.
7. System V Interface Definition (SVID). AT&T Customer Information Center, Attn: Customer Service Representative, PO Box 19901, Indianapolis, IN 46219, USA. Issue 2, 1986, ISBN 0-932764-10-X.
8. POSIX Explored. /usr/group technical publication. /usr/group, 4655 Old Ironsides Drive #200, Santa Clara, CA 95054, USA. Published 1987, ISBN 0-936593-05-9.
9. POSIX Comparison with MORE/bsd. Deborah Scherrer, Mt Xinu. January 6, 1988.

SunOS Virtual Memory Implementation

Joseph P. Moran

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043 USA

ABSTRACT

The implementation of a new virtual memory (VM) system for Sun's implementation of the UNIX operating system (SunOS‡) is described. The new VM system was designed for extensibility and portability using an object-oriented design carefully constructed to not compromise efficiency. The basic implementation abstractions of the new VM system and how they are managed are described. Some of the more interesting problems encountered with a system based on mapped objects and the resolution taken to these problems are described.

1. Introduction

In December 1985 our group at Sun Microsystems began a project to replace our 4.2BSD-based VM system with a VM system engineered for the future. A companion paper [1] describes the general architecture of our new VM system, its goals, and its design rationale. To summarize, this architecture provides:

- Address spaces that are described by mapped objects.
- Support for shared or private (copy-on-write) mappings.
- Support for large, sparse address spaces.
- Page level mapping control.

We wanted the new VM system's implementation to reflect the clean design of its architecture and felt that basing the implementation itself on the proper set of abstractions would result in a system that would be efficient, extensible to solving future problems, readily portable to other hardware architectures, and understandable. Our group's earlier experience in implementing the *vnode* architecture [2] had shown us the utility of using object-oriented programming techniques as a way of devising useful and efficient implementation abstractions, so we chose to apply these techniques to our VM implementation as well.

The rest of this paper is structured as follows. Section 2 provides an overview of the basic object types that form the foundation of the implementation, and sections 3 through 6 describe these object types in detail. Sections 7 through 9 describe related changes made to the rest of the SunOS kernel. The most extensive changes were those related to the file system object managers. A particular file system type is used to illustrate those changes. Section 10 compares the performance of the old and new VM implementations. Sections 11 and 12 discuss conclusions and plans for future work.

2. Implementation Structure

The initial problem we faced in designing the new VM system's implementation was finding a clean set of implementation abstractions. The system's architecture suggested some candidate abstractions and examining the architecture with an eye toward carving it into a collection of objects suggested others.

We ultimately chose the following set of basic abstractions.

- The architecture allows for page-level granularity in establishing mappings from file system objects to virtual addresses. Thus the implementation uses the *page* structure to keep track of information about physical memory pages. The object managers and the VM system use this data structure to manage physical memory as a cache.

‡ SunOS is a trademark of Sun Microsystems.

- The architecture defines the notion of an ''address space''. In the implementation, an *address space* consists of an ordered linked list of mappings. This level defines the external interface to the VM system and supplies a simple procedural interface to its primary client, the UNIX kernel.
- A *segment* describes a contiguous mapping of virtual addresses onto some underlying entity. The corresponding layer of the implementation treats segments as objects, acting as a class in the C++ [3] sense[1]. Segments can map several different kinds of target entities. The most common mappings are to objects that appear in the file system name space, such as files or frame buffers. Regardless of mapping type, the segment layer supplies a common interface to the rest of the implementation. Since there are several types of segment mappings, the implementation uses different *segment drivers* for each. These drivers behave as subclasses of the segment class.
- The *hardware address translation* (*hat*) layer is the machine dependent code that manages hardware translations to pages in the machine's memory management unit (MMU).

The VM implementation requires services from the rest of the kernel. In particular, it makes heavy demands of the *vnode* [2] object manager. The implementation expects the *vnode* drivers to mediate access to pages comprising file objects. The part of the *vnode* interface dealing with cache management changed drastically. Finding the right division of responsibility between the segment layer and the *vnode* layer proved to be unexpectedly difficult and accounted for much of the overall implementation effort.

The new VM system proper has no knowledge of UNIX semantics. The SunOS kernel provides UNIX semantics by using the VM abstractions as primitive operations [1]. Figure 1 is a schematic diagram of the VM abstractions and how they interact. The following sections describe in more detail the implementation abstractions summarized above.

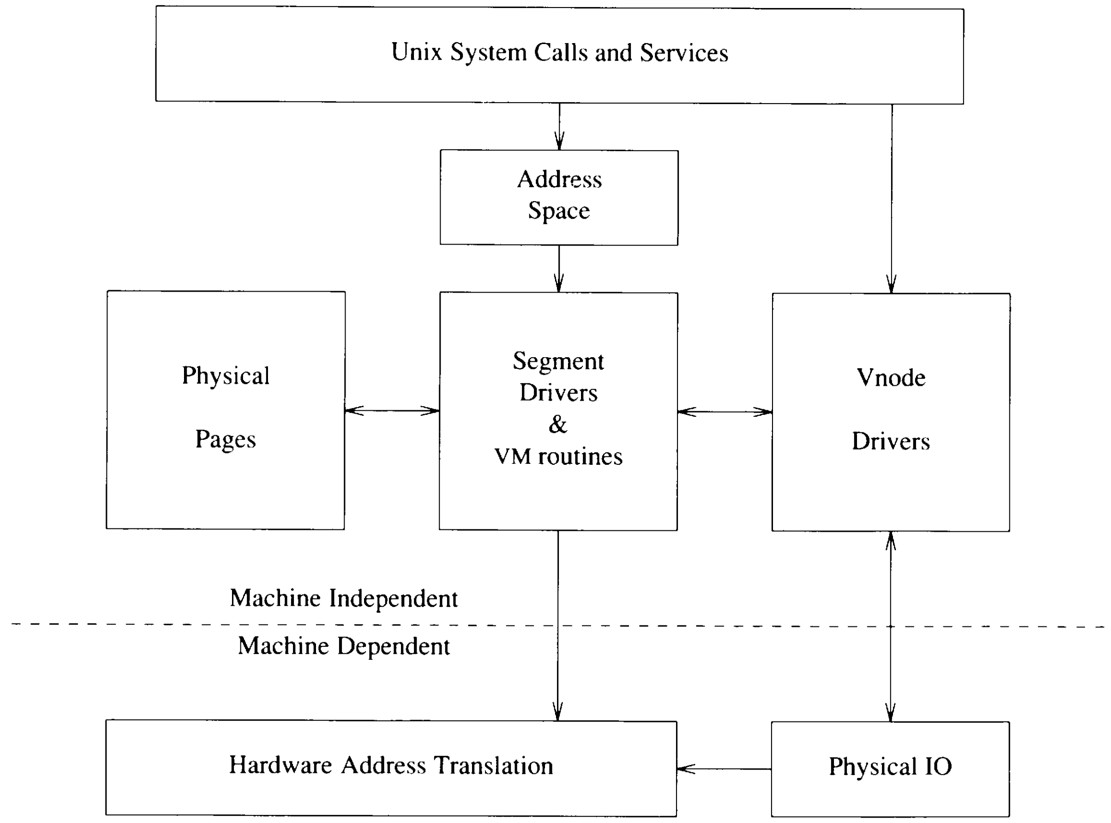

Unix System Calls and Services
Address Space
Physical Pages
Segment Drivers & VM routines
Vnode Drivers
Machine Independent
Machine Dependent
Hardware Address Translation
Physical IO

Figure 1

[1] Actually, as a class whose public fields are all virtual, so that subclasses are expected to define them.

3. *page* Structure

The new VM architecture treats physical memory as a cache for the contents of memory objects. The *page* is the data structure that contains the information that the VM system and object managers need to manage this cache. The *page* structure maintains the identity and status of each page of physical memory in the system. There is one *page* structure for every interesting[2] page in the system.

A *page* represents a system page size unit of memory that is a multiple of the hardware page size. The memory page is identified by a <*vnode*, offset> pair kept in the *page* structure. Each page with an identity is initialized to the contents of a page's worth of the *vnode*'s data starting at the given byte offset. A hashed lookup based on the <*vnode*, offset> pair naming the page is used to find a page with a particular name. The implementation keeps all pages for a given *vnode* on a doubly-linked list rooted at the *vnode*. Maintaining this list speeds operations that need to find all a *vnode*'s cached pages. *page* structures can also be on free lists or on an ''I/O'' list depending on the setting of page status flags. The *page* structure also contains an opaque pointer that the *hat* layer uses to maintain a list of all the active translations to the page that are loaded in the hardware. In the machine independent VM code above the *hat* layer, the only use for this opaque pointer is to test for NULL to determine if there are any active translations to the page. When the machine-dependent *hat* layer unloads a translation it retrieves the hardware reference and modified bits for that translation to the page, and merges them into machine-independent versions of these bits maintained in the *page* structure.

4. Address Space

The highest level abstraction that the VM system implements is called an *address space* (*as*), which consists of a collection of mappings from virtual addresses to underlying objects such as files and display device frame buffers. The *as* layer supports a procedural interface whose operations fall into two basic classes. Procedures in the first class manipulate an entire address space and handle address space allocation, destruction, duplication, and ''swap out''. Procedures in the second class manipulate a virtual address range within an address space. These functions handle fault processing, setting and verifying protections, resynchronizing the contents of an address space with the underlying objects, obtaining attributes of the mapped objects, and mapping and unmapping objects. Further information on these functions may be found in [1].

The implementation must maintain state information for each address space. The heart of this information is a doubly linked list of contiguous mappings (termed *segments* for lack of a better name) sorted by virtual address. Section 5 describes segments in detail. The *as* layer implements its procedural interface by iterating over the required range of virtual addresses and calling the appropriate segment operations as needed.

In addition, the *as* structure contains a *hardware address translation* (*hat*) structure used to maintain implementation specific memory management information. Positioning the *hat* structure within the *as* structure allows the machine dependent *hat* layer to describe all the physical MMU mappings for an address space, while the machine independent *as* layer manages all the virtual address space mappings. The *hat* structure is opaque to the machine independent parts of the system and only the *hat* layer examines it. Section 6 describes the *hat* layer in detail. The *as* structure also includes machine independent address space statistics that are kept separately from the machine dependent *hat* structure for convenience.

4.1. Address Space Management

The implementation uses several techniques to reduce the overhead of *as* management. To reduce the time to find the segment for a virtual address, it maintains a ''hint'' naming the last segment found, in a manner similar to the technique used in Mach [4]. Any time the *as* layer translates a virtual address to a segment, this hint is used as the starting point to begin the search.

Another optimization reduces the total number of segments in a given address space by allowing segment drivers to coalesce adjacent segments of similar types. This reduces the average time to find the segment that maps a given virtual address within an address space. By using this technique, the common UNIX *brk*(2) system call normally reduces to a simple segment extension within the process address space.

[2] Pages for kernel text and data and for frame buffers are not considered ''interesting''.

4.2. Address Space Usage

SunOS uses an *as* to describe the kernel's own address space, which is shared by all UNIX processes when operating in privileged (supervisor) mode. A UNIX process typically has an *as* to describe the address space it operates in when in non-privileged (user) mode[3]. An *as* is an abstraction that exists independent of any of its uses. Just as several UNIX processes share the same kernel address space when operating in supervisor mode, an *as* can have multiple threads of control active in an a user mode address space at the same time. Future implementations of the operating system will take advantage of these facilities [5].

Most UNIX memory management system calls map cleanly to calls on the *as* layer. The *as* layer does not have knowledge of the implementation of the segment drivers below it, thus making it easy to add new segment types to the system. The *as* design provides support for large sparse address spaces without undue penalty for common cases, an important consideration for the future software demands that will be placed on the VM system.

5. Segments

A *segment* is a region of virtual memory mapped to a contiguous region of a memory object[4]. Each segment contains some public and private data and is manipulated in an object-oriented fashion. The public data includes the base and size of the segment in page-aligned bytes, pointers for the next and previous segments in the address space, and a pointer to the *as* structure itself. Each segment also contains a reference to a vector of pointers to operations (an ''ops'' vector) that implement a set of functions similar to the *as* functions, and a pointer to a private per-segment type data structure. This is similar to the way the SunOS *vnode* and *vfs* abstractions are implemented [2]. Using this style of interface allows multiple segment types to be implemented without affecting the rest of the system.

To most efficiently handle its data structures, a segment driver is free to coalesce adjacent segments of the same type in the virtual address space or even to break a segment down into smaller segments. Individual virtual pages within a segment's mappings may have varying attributes (e.g. protections). This design allows the segment abstraction control over the attributes and data structures it manages.

Of equal importance to what a segment driver does is what it does not do. In particular, we found that having the segment driver handle the page lookup operation and call the *vnode* object manager only when a needed page cannot be found was a bad idea. After running into some problems that could not be solved as a result of this split, we restructured the VM system so that the segment driver always asks the object manager for the needed page on each fault. Having the *vnode* object manager be responsible for the page lookup operation allows it to take action on each new reference.

5.1. Segment Driver Types

The implementation includes the following segment driver types:

seg_vn	Mappings to regular files and anonymous memory.
seg_map	Kernel only transient <*vnode*, offset> translation cache.
seg_dev	Mappings to character special files for devices (e.g. frame buffers).
seg_kmem	Kernel only driver used for miscellaneous mappings.

The seg_vn and seg_map segment drivers manage access to *vnode* memory objects and are the primary segment drivers.

5.2. *vnode* Segment

The **seg_vn** *vnode* segment driver provides mappings to regular files. It is the most heavily used segment driver in the system.

The arguments to the segment create function include the *vnode* being mapped, the starting offset, the mapping type, the current page protections, and the maximum page protections. The mapping type can be shared or private (copy-on-write). With a shared mapping, a successful memory write access to the mapped region will cause the underlying file object to be changed. With a private mapping the first write access to a page of the mapped region will cause a copy-on-write operation that creates a private page and initializes it to a copy of the original page.

[3] Some processes run entirely in the kernel and have no need for a user mode address space.

[4] Note that the name ''segment'' is not related to traditional UNIX text, data, and stack segments.

The UNIX *mmap*(2) system call, which sets up new mappings in the process's user address space, calculates the maximum page protection value for a shared mapping based on the permissions granted on the *open* of the file. Thus, the *vnode* segment driver will not allow a file to be modified through a mapping if the file was originally opened read-only.

5.2.1. Anonymous Memory

An important aspect of the VM system is the management of ''anonymous'' pages that have no permanent backing store. An anonymous page is created for each copy-on-write operation and for each initial fault to the anonymous clone object[5]. For a UNIX executable, the uninitialized data and stack are set up as private mappings to the anonymous clone object.

The mechanism used to manage anonymous pages has been isolated to a set of routines that provide a service to the rest of the VM system. Segment drivers that choose to implement private mappings use this service. The *vnode* segment driver is the primary user of anonymous memory objects.

5.2.1.1. Anonymous Memory Data Structures

The *anon* structure serves as a name for each active anonymous page of memory. This structure introduces a level of indirection for access to anonymous pages. We do not wish to assume that anonymous pages can be named by their position in a storage device, since we would like to be able to have anonymous pages in memory that haven't been allocated swap space. The *anon* data structure is opaque above the anonymous memory service routines and is operated on using a procedural interface in an object-oriented fashion. These objects are reference counted, since there can be more than one reference to an anonymous page[6]. This reference counting allows the *anon* procedures to easily detect when an anonymous page and corresponding resident physical page (if any) are no longer needed.

The other data structure related to anonymous memory management is the *anon_map* structure. This structure describes a cluster of anonymous pages as a unit. The *anon_map* structure consists of an array of *anon* structure pointers with one *anon* pointer per page. Segment drivers that wish to refer to anonymous pages do so by using an *anon_map* structure to keep an array of pointers to *anon* structures for the anonymous pages. These segment drivers lazily allocate an *anon_map* structure with NULL *anon* structure pointers at fault time as needed (i.e., on the first copy-on-write for the segment or on the first fault for an all anonymous mapping).

5.2.1.2. Anonymous Memory Procedures

There are two *anon* procedures that operate on the arrays of *anon* structure pointers in the *anon_map* structure. *anon_dup()* copies from one *anon* pointer array to another one, incrementing the reference count on every allocated *anon* structure. This operation is used when a private mapping involving anonymous memory is duplicated. The converse of *anon_dup()* is *anon_free()*, which decrements the reference count on every allocated *anon* structure. If a reference count goes to zero, the *anon* structure and associated page are freed. *anon_free()* is used when part of a privately mapped anonymous memory object is unmapped.

There are three *anon* procedures used by the fault handlers for anonymous memory objects. *anon_private()* allocates an anonymous page, initializing it to the contents of the previous page loaded in the MMU. *anon_zero()* is similar to *anon_private()*, but initializes the anonymous page to zeroes. This routine exists as an optimization to avoid having to copy a page of zeroes with *anon_private()*. Finally, *anon_getpage()* retrieves an anonymous page given an *anon* structure pointer.

5.2.2. *vnode* Segment Fault Handling

Page fault handling is a central part of the new VM system. The fault handling code resolves both hardware faults (e.g., hardware translation not valid or protection violation) and software pseudo-faults (e.g., lock down pages). The *as* fault handing routine is called with a virtual address range, the fault type (e.g., invalid translation or protection violation), and the type of attempted access (read, write, execute). It performs a segment lookup operation based on the virtual address and dispatches to the segment driver's fault routine, which is responsible for resolving the fault.

[5] The name of this object in the UNIX file system name space is **/dev/zero**.

[6] Typically from an address space duplication resulting from a UNIX *fork*(2) system call.

The *vnode* segment driver takes the following steps to handle a fault.

- Verify page protections.
- If needed, allocate an *anon_map* structure.
- If needed, get the page from the object manager.
  Call the *hat* layer to load a translation to the page.
- If needed, obtain a new page by performing copy-on-write.
  Call the *hat* layer to load a writable translation to the new page.

Some specific examples of *vnode* segment fault handling and how anonymous memory is used are given below.

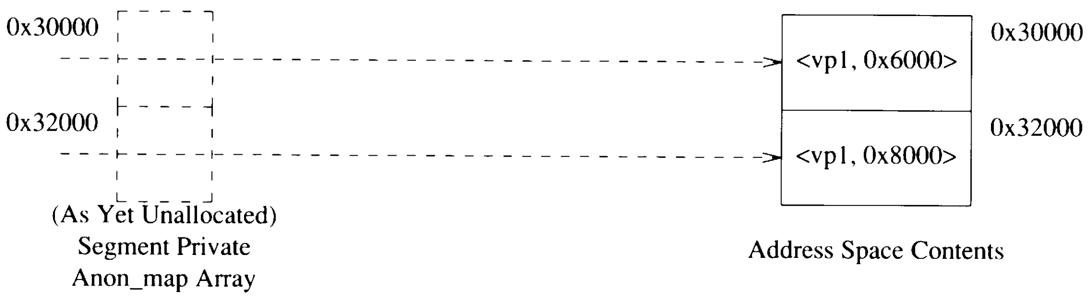


Figure 2
<vp1, 6000> Mapped Private to Address 0x30000 for 0x4000 Bytes

Figure 2 depicts a private mapping from offset 0x6000 in *vnode* vp1 to address 0x30000 for a length of 0x4000 bytes using a system page size of 0x2000. If a non-write fault occurs on an address within the segment, the *vnode* segment driver asks the *vnode* object manager for the page named by <vp1, 0x6000 + (*addr* - 0x30000)>. The *vnode* object manager is responsible for creating and initializing the page when requested to do so by a segment driver. After obtaining the page, the *vnode* segment driver calls the *hat* layer to load a translation to the page. The permissions passed to the *hat* layer from the *vnode* segment driver are for a read-only translation since this is a private mapping for which we want to catch a memory write operation to initiate a copy-on-write operation.

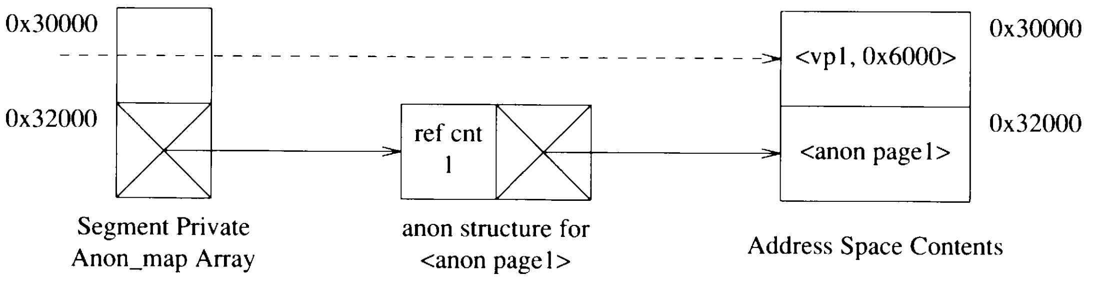


Figure 3
After Copy-On-Write Operation to Address 0x32000

Figure 3 shows the results of a copy-on-write operation on address 0x32000 in Figure 2. The *vnode* segment driver has allocated an *anon_map* structure and initialized the second entry to point to an allocated *anon* structure that initially has a reference count of one. The *anon_private()* routine has allocated the *anon* structure in the array, returned a page named by that *anon* structure, and initialized to the contents of the previous page at 0x32000. After getting the anonymous page from *anon private()*, the *vnode* segment driver calls the *hat* layer to load a writable translation to the newly allocated and initialized page.

Note that as an optimization, the *vnode* segment driver is able to perform a copy-on-write operation, even if the original translation was invalid, since the fault handler gets a fault type parameter (read, write, execute). If the first fault taken in the *segment* described in Figure 3 is a write fault at address 0x32000 then the first operation is to obtain the page for <vp1, 0x8000> and call the *hat* layer to load a read-only translation. The *vnode* segment driver can then detect that it still needs to perform the copy-on-write operation because the fault type was for a write access. If the copy-on-write operation is needed, the *vnode* segment driver will call *anon_private()* to create a private copy of the page.

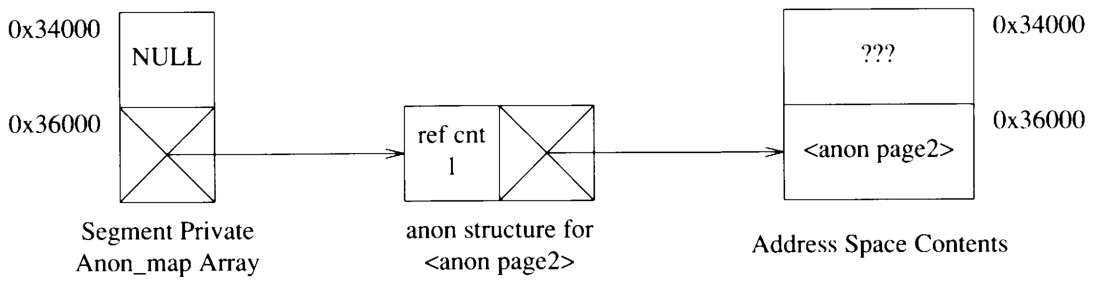


Figure 4
Private Mapping to /dev/zero for 0x4000 bytes at Address 0x34000
After a Page Fault at Address 0x36000

Figure 4 depicts a private mapping from the anonymous clone device **/dev/zero** to address 0x34000 for length 0x4000 after a page fault at address 0x36000. Since there is no primary *vnode* that was mapped, the *vnode* segment driver calls *anon_zero()* to allocate an *anon* structure and corresponding page and initialize the page to zeroes.

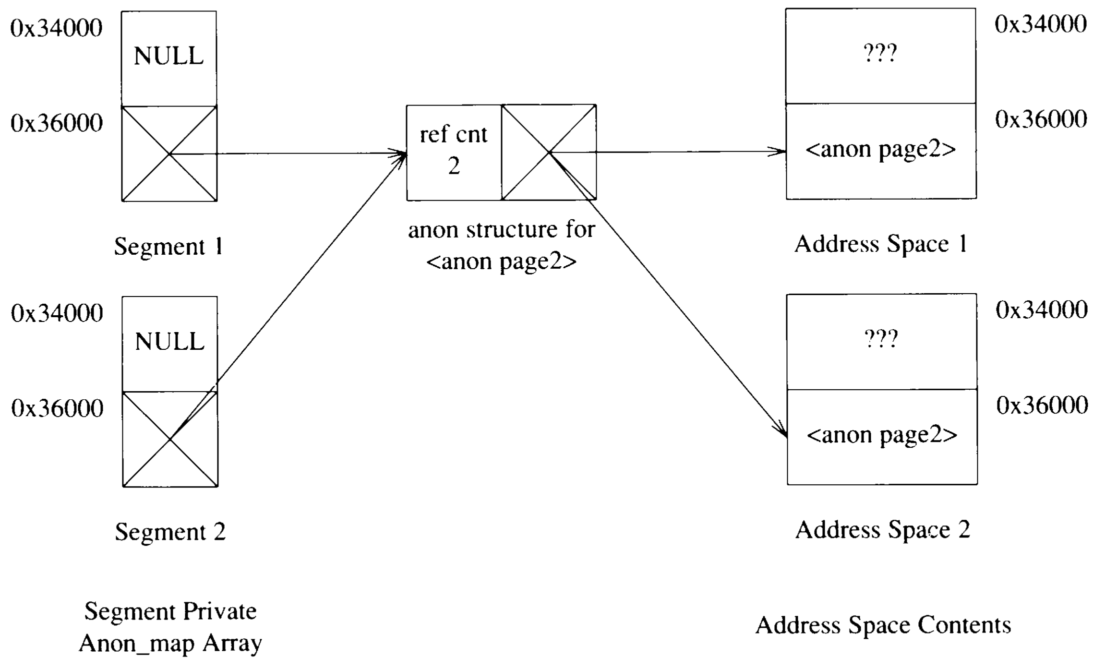


Figure 5
After the Private Mapped Vnode Segment is Duplicated

Figure 5 shows what happens when the mapped private *vnode* segment shown in Figure 4 is duplicated. Here both segments have a private reference to the same anonymous page. When the segment is duplicated, *anon_dup()* is called to increment the reference count on all the segment's allocated *anon* structures. In this example, there is only one allocated *anon* structure and its reference count has been incremented from one to two. Also, as part of the *vnode* segment duplication process for a privately mapped segment, all the *hat* translations are changed to be read-only so that previously writable anonymous pages are now set up for copy-on-write.

Figure 6 shows the result after the duplicated segment in Figure 5 handles a write fault at address 0x36000. When the segment fault handler calls *anon_getpage()* to return the page for the given *anon* structure, it will return protections that force a read-only translation since the reference count on the *anon* structure is

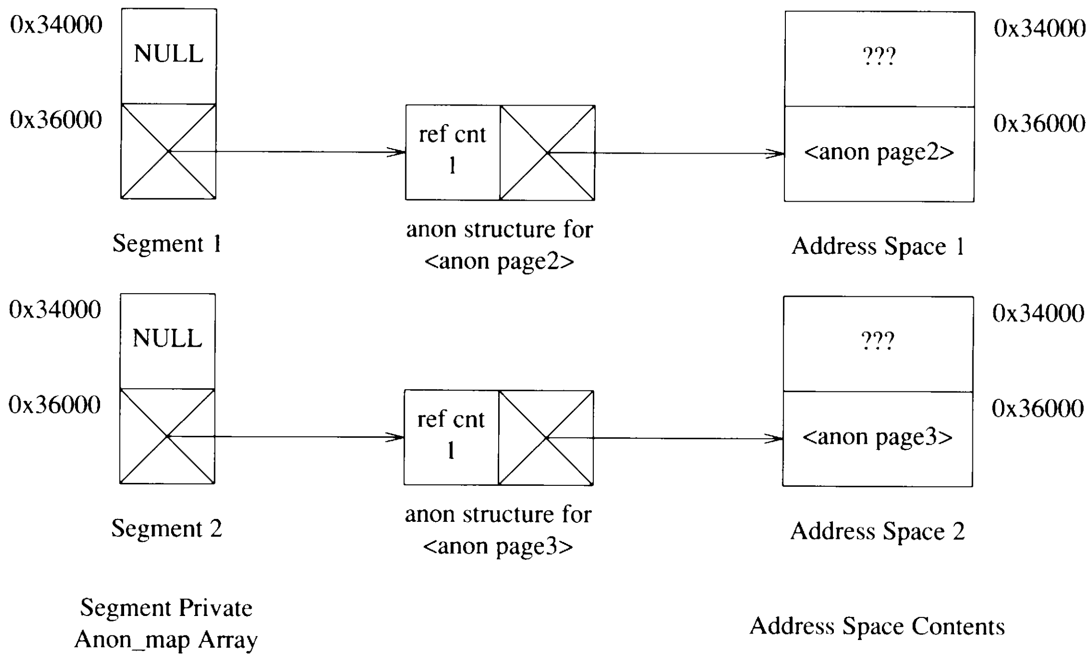


Figure 6
After Write Fault on Address 0x36000 in Address Space 2

greater than one. The segment driver fault handler will then call *anon_private()* to allocate a new *anon* structure and *page* structure and to initialize the page to the contents of the previous page loaded in the MMU. In contrast to the case depicted in Figure 3, *anon_private()* is copying from another anonymous page and will decrement the reference count of the old *anon* structure after the *anon* pointer in the segment's *anon_map* array is changed to point to the newly allocated *anon* structure. Since the reference count on the original *anon* structure reverts to one, this means that the original segment will no longer have to do a copy-on-write operation for a subsequent write fault at address 0x36000. If a fault were to occur at 0x36000 in the original segment, *anon_getpage()* would not enforce a read-only mapping, since the reference count for the *anon* structure is now one.

5.3. Kernel Transient *vnode* Mapping Segment

The **seg_map** segment driver is a driver the kernel uses to get transient <*vnode*, offset> mappings. It supports only shared mappings. The most important service it provides to the *as* layer is fault resolution for kernel page faults. The seg_map driver manages a large window of kernel virtual space and provides a view onto a varying subset of the system's pages. The seg_map driver manages its own virtual space as a cache, so that recently referenced <*vnode*, offset> pairs are likely to be loaded in the MMU and no page fault will be taken when the virtual address within the seg_map segment are referenced.

This segment driver provides fast map and unmap operations using two segment driver-specific subroutines: *segmap_getmap()* and *segmap_release()*. Given a <*vnode*, offset> pair, *segmap_getmap()* returns a virtual address within the seg_map segment that is initialized to map part of the *vnode*. This is similar to the traditional UNIX *bread()* function used in the "block IO system" to obtain a buffer that contains some data from a block device. The *segmap_release()* function takes a virtual address returned from *segmap_getmap()* and handles releasing the mapping. *segmap_release()* also handles writing back modified pages. *segmap_release()* performs a similar function to the traditional UNIX *brelse()* / *bdwrite()* / *bwrite()* / *bawrite()* "block IO system" procedures depending on the flags given to *segmap_release()*.

The seg_map driver is simply used as an optimization in the kernel over the standard *vnode* driver. It is important to be able to do fast map and unmap operations in the kernel to implement *read*(2) and *write*(2) system calls. The basic algorithm for the *vnode* read and write routines is to get a mapping to the file, copy the data from/to the mapping, and then unmap the file. Note that the kernel accesses the file data just as user processes do by using a mapping to the file. The *vnode* routines that implement read and write use *segmap_getmap()* and *segmap_release()* to provide the fast map and unmap operations within the kernel's address space.

5.4. Device Driver Segment

The **seg_dev** segment driver manages objects controlled by character special ("raw") device drivers that provide an *mmap* interface. The most common use of the seg_dev driver is for mapped frame buffers, though it is also used for mappings to machine-specific memory files such as physical memory, kernel virtual memory, Multibus memory, or VMEbus memory. This driver currently only supports shared mappings and does not deal with anonymous private memory pages. The driver is simple since it doesn't have to worry about a many operations that don't make sense for these types of objects (e.g., swap out). To resolve a fault, it simply calls a function to return an opaque "cookie" from the device driver, which is then handed to the machine-specific *hat* layer to load a translation to the physical page denoted by the cookie.

5.5. Kernel Memory Segment

The **seg_kmem** segment driver is an example of the use of a machine independent concept to solve a machine dependent problem. The kernel's address space is described by an *as* structure just like the user's address space. The seg_kmem segment driver is used as a catch-all to map miscellaneous entities into the kernel's address space. These entities includes the kernel's text, data, bss, and dynamically allocated memory space. This driver also manages other machine dependent portions of the kernel's address space (e.g. Sun's Direct Virtual Memory Access space [6]).

The seg_kmem driver currently only supports non-paged memory whose MMU translations are always locked[7]. In the previous 4.2BSD-based VM system, the management of the kernel's address space for things like device registers was done by calls to a *mapin()* procedure that set up MMU translations using a machine-dependent page table entry. For kernel and driver compatibility reasons, the seg_kmem driver supports a *mapin*-like interface as a set of segment driver-specific procedures.

6. Hardware Address Translation Layer

The *hardware address translation* (*hat*) layer is responsible for managing the machine dependent memory management unit. It provides the interface between the machine dependent and the machine independent parts of the VM system. The machine independent code above the *hat* layer knows nothing about the implementation details below the *hat* layer. The clean separation of machine independent and dependent layers of the VM system allows for better understandability and faster porting to new machines with different MMUs.

The *hat* layer exports a set of procedures for use by the machine independent segment drivers. The higher levels cannot look at the current mappings, they can only determine if any mappings exist for a given page. The machine independent levels call down to the *hat* layer to set up translations as needed. The basic dependency here is the ability to handle and recover from page faults (including copy-on-write). The *hat* layer is free to remove translations as it sees fit if the translation was not set up to be locked. There exists a call back mechanism from the hat layer to the segment driver so that the virtual reference and modified bits can be maintained when a translation is take away by the *hat* layer. This ability is needed for alternate paging techniques in which per address space management of the working set is done.

6.1. *hat* Procedures

Table 1 lists the machine independent *hat* interfaces. All these procedures must be provided, although they may not necessarily do anything if not required by the *hat* implementation for a given machine.

6.2. *hat* Implementations

Several *hat* implementations have already been completed. The first implementations were for the Sun MMU [6]. The MMUs in the current Sun-2/3/4 family are quite similar. All use a fixed number of context, segment, and page table registers in special hardware registers to provide mapping control. The Sun-2 MMU has a separate context when running in supervisor mode whereas the Sun-3 and Sun-4 MMUs have the kernel mapped in each user context. The maximum virtual address space for the Sun-2, Sun-3, and Sun-4 MMUs are 16 megabytes, 256 megabytes, and 4 gigabytes respectively.

Some machines in the Sun-3 and Sun-4 families use a virtual address write-back cache. The use of a virtual address cache allows for faster memory access time on cache hits, but can be a cause of great

[7] This means that the *hat* layer cannot remove any of these translations without explicitly being told to do so by the seg_kmem driver.

Operation	Function
`hat_init()`	One time *hat* initialization.
`hat_alloc(as)`	Allocate *hat* structure for *as*.
`hat_free(as)`	Release all *hat* resources for *as*.
`hat_pageunload(pp)`	Unload all translations to page *pp*.
`hat_pagesync(pp)`	Sync ref and mod bits to page *pp*.
`hat_unlock(seg, addr)`	Unlock translation at *addr*.
`hat_chgprot(seg, addr, len, prot)`	Change protection values.
`hat_unload(seg, addr, len)`	Unload translations.
`hat_memload(seg, addr, pp, prot, flags)`	Load translation to page *pp*.
`hat_devload(seg, addr, pf, prot, flags)`	Load translation to cookie *pf*.

Table 1
***hat* operations**

trouble to the kernel in the old VM system [7]. Since the *hat* layer has information about all the translations to a given page, it can manage all the details of the virtual address cache. It can verify the current virtual address alignment for the page and decide to trade translations if an attempt to load a non-cache consistent address occurs. In the old 4.2BSD-based VM system the additional support needed for the virtual address cache permeated many areas of the system. Under the new VM system, support for the virtual address cache is isolated within the *hat* layer.

Other *hat* implementations have been done for more traditional page table-based systems. The Motorola 68030 has a flexible on-chip MMU. The *hat* layer chooses to manage it using a three level page table to support mapping a large sparse virtual address space with minimal overhead. The Intel 80386 also has an on-chip MMU, but it has a fixed two level translation scheme of 4KB pages. The problem with the 80386 MMU is that the kernel can write all pages regardless of the page protections (i.e., the write protection only applies to non-supervisor mode accesses)! This means that explicit checks must be performed for kernel write accesses to read-only pages so that kernel protection faults can be simulated. Another implementation has been done for IBM 370/XA compatible main frames. The biggest problem with this machine's architecture for the new VM system is that an attempted write access to a read-only page causes a protection exception that can leave the machine in an unpredictable state for certain instructions that modify registers as a side effect. These instructions cannot be reliably restarted thus breaking copy-on-write fault handling. The implementation resorts to special work arounds for the few instructions that exhibit this problem[8].

7. File System Changes

The VM system required changes to several other parts of the SunOS kernel. The VM system relies heavily on the *vnode* object managers, and required changes to the *vnode* interface as well as to each *vnode* object type implementation. It took us several attempts to get the new *vnode* interface right.

Our initial attempt gave the core VM code responsibility for all decisions about operations it initiated. We repeatedly encountered problems induced by not having appropriate information available within the VM code at sites where it had to make decisions, and realized that the proper approach was to make decisions at locations possessing the requisite information. The primary effect of this shift in responsibility was to give the *vnode* drivers control on each page reference. This allows the *vnode* drivers to recognize and act on each new reference. These actions include validating the page, handling any needed backing store allocation, starting read-ahead operations, and updating file attributes.

7.1. File Caching

Traditionally, buffers in the UNIX buffer cache have been described by a device number and a physical block number on that device. This use of physical layout information requires all file system types implemented on top of a a block device to translate (*bmap*) each logical block to a physical block on the

[8] Such instructions are highly specialized and the standard compilers never generate them.

device before it can be looked up in the buffer cache.

In the new VM system, the physical memory in the system is used as a *logical* cache; each buffer (page) in the cache is described by an object name (*vnode*) and a (page-aligned) offset within that object. Each file is named as a separate *vnode*, so the VM system need not have any knowledge of the way the *vnode* object manager (file system type) stores the *vnode*. A segment driver simply asks the *vnode* object manager for a range of logical pages within the *vnode* being mapped. The file system independent code in the segment drivers only has to deal with offsets into a *vnode* and does not have to maintain any file system-specific mapping information that is already kept in the file system-specific data structures. This provides a much cleaner separation between the segment and *vnode* abstractions and puts few constraints on the implementation of a *vnode* object manager[9].

The smallest mapping unit relevant to the VM system is a system page. However, the system page size is not necessarily related to the block sizes that a file system implementation might use. While we could have implemented a new file system type that used blocks that were the same size as a system page, and only supported file systems that had this attribute, we did not feel this was an acceptable approach. We needed to support existing file systems with widely varying block size. We also did not feel that it was appropriate to use only one system page size across a large range of machines of varying memory size and performance. We decided it was best to push the handling of block size issues into each file system implementation, since the issues would vary greatly depending on the file system type.

The smallest allocatable file system block is potentially smaller than the system page size, while the largest file system block may be much larger than the system page size. The *vnode* object manager must initialize each page for a file to the proper contents. It may do this by reading a single block, multiple blocks, or possibly part of a block, as necessary. If the size of the file is not a multiple of the system page size, the *vnode* object manager must handle zeroing the remainder of the page past the end of the file.

Using a logical cache doesn't come without some cost. When trying to write a page back to the file system device, the VOP_PUTPAGE routine (discussed below) may need to map the logical page number within the object to a physical block number, or perhaps to a list of physical block numbers. If the file system-specific information needed to perform the mapping function is not present in memory, then a read operation may be required to get it. This complicates the work the page daemon must do when writing back a dirty page. File system implementations need to be careful to prevent the page daemon from deadlocking waiting to allocate a page needed for a *bmap*-like operation while trying to push out a dirty page when there are no free pages available.

7.2. *vnode* Interface Changes

We defined three new *vnode* operations for dealing with the new abstractions of mappings in address spaces and pages. These new *vnode* operations replaced ones that dealt with the old buffer cache and the 4.2BSD-based VM system [2]. The primary responsibility of the *vnode* page operations is to fill and drain physical pages (page-in and page-out). It also provides an opportunity for the managers of particular objects to map the page abstractions to the representation used by the object being mapped.

The VOP_MAP() routine is used by the *mmap* system call and is responsible for handling file system dependent argument checking, as well as setting up the requested mapping. After checking parameters it uses two address space operations to do most of the work. Any mappings in the address range specified in the *mmap* system call are first removed by using the *as_unmap()* routine. Then the *as_map()* routine establishes the new mapping in the given address space by calling the segment driver selected by the *vnode* object manager.

The VOP_GETPAGE() routine is responsible for returning a list of pages from a range of a *vnode*. It typically performs a page lookup operation to see if the pages are in memory. If the desired pages are not present, the routine does everything needed to read them in and initialize them. It has the opportunity to perform operations appropriate to the underlying *vnode* object on each fault, such as updating the reference time or performing validity checks on cached pages.

As an optimization, the VOP_GETPAGE() routine can return extra pages in addition to the ones requested. This is appropriate when a physical read operation is needed to initialize the pages and the *vnode* object manager tries to perform the I/O operation using a size optimal for the particular object. Before this is done the segment driver is consulted, using a ''kluster'' segment function, so that the segment driver has

9 We have taken advantage of this and have implemented several interesting *vnode* object managers that are nothing like typical file systems.

the opportunity to influence the *vnode* object manager's decisions. The VOP_GETPAGE() routine also handles read-ahead if it detects a sequential access pattern on the *vnode*. It uses the same segment kluster function to verify that the segment driver believes that it would be worthwhile to perform the read-ahead operation. The I/O klustering and read-ahead conditions allow both the *vnode* object manager and the segment driver controlling a mapping onto this object to have control over how these conditions are handled. Thus, for these conditions we have set up our object-oriented interfaces to allow distributed control among different abstractions that have different pieces of knowledge about a particular problem. The *vnode* object manager has knowledge about preferred I/O size and reference patterns to the underlying object, whereas the segment driver has the knowledge about the view established to this object and may have advice passed in from above the address space regarding the expected reference pattern to the virtual address space.

The other new *vnode* operation for page management is VOP_PUTPAGE(). This operation is the complement of VOP_GETPAGE() and handles writing back potentially dirty pages. A flags parameter controls whether the write back operation is performed asynchronously and whether the pages should be invalidated after being written back.

The VOP_GETPAGE() and VOP_PUTPAGE() interfaces deal with offsets and pages in the logical file. No information about the physical layout of the file is visible above the *vnode* interface. This means that the work of translating from logical blocks to physical disk blocks (the *bmap* function) is all done within the *vnode* routines that implement the VOP_GETPAGE() and VOP_PUTPAGE() interfaces. This is a clean and logical separation of the file object abstractions from the VM abstractions and contrasts with the old 4.2BSD-based implementation where the physical locations of file system blocks appeared in VM data structures.

8. UFS File System Rework

Another difficult issue pertinent to the conversion to a memory-mapped, page-based system is how to convert existing file systems. The most notable of these in SunOS is the 4.2BSD file system [8], which is known in SunOS as the UNIX File System (UFS). The relevant characteristics of this file system type include support for two different blocking sizes (a large basic block size for speed, and a smaller fragment size to avoid excessive disk waste), the ability to have unallocated blocks (''holes'') in the middle of a file which read back as zeroes, and the need to *bmap* from logical blocks in the file to physical disk blocks.

8.1. Sparse UFS File Management

ufs_getpage() is the UFS routine that implements the VOP_GETPAGE() interface. When a fault occurs on a UFS file, the segment driver fault routine calls this routine, passing it the type of the attempted access (e.g., read or write access). It uses this access type information to determine what to do if the requested page corresponds to an as yet unallocated section of a sparse file. If a write access to one of these holes in the file is attempted, *ufs_getpage()* will attempt to allocate the needed block(s) of file system storage. If the allocation fails because there is no more space available in the file system, or the user process has exceeded its disk quota limit, *ufs_getpage()* returns the error back to the calling procedure which then propagates back to the caller of address space fault routine.

When *ufs_getpage()* handles a read access to a page that does not have all its disk blocks allocated, it zeroes out the part of the page that is not backed by an allocated disk block and arranges for the segment driver requesting the page to establish a read-only translation to it. Thus no allocation is done when a process tries to read a hole from a UFS file. However, an attempted write access to such a page causes a protection fault and *ufs_getpage()* can perform the needed file system block allocation as previously described.

8.2. UFS File Times

Another set of problems resulted from handling the file access and modified times. The obvious way to handle this problem is to simply update the access time in *ufs_getpage()* any time a page is requested and to update the modification time in *ufs_putpage()* any time a dirty page is written back. However, this approach has some problems.

The first problem is that the UFS implementation has never marked the access time when doing a *write* to the file[10]. The second problem is related to getting the correct modification time when writing a file.

[10] The ''read'' that is sometimes needed to perform a *write* operation never causes the file's access time to be updated.

When doing a *write*(2) system call, the file is marked with the current time. When dirty pages created by the *write* operation are actually pushed back to backing store in *ufs_putpage(),* we don't want to override the modification time already stored in the *inode*[11].

To solve these problems, *inode* flags are set in the "upper layers" of the UFS code (e.g., when doing a file read or write operation) and examined in the "lower layers" of the UFS code (*ufs_getpage*() and *ufs_putpage*()). *ufs_getpage()* examines the *inode* flags to determine whether to update the *inode*'s access time based on whether a read or write operation is currently in progress. *ufs_putpage()* can use the *inode* flags to determine whether it needs to update the *inode*'s modification time based on whether the modification time has been set since the last time the *inode* was written back to disk.

8.3. UFS Control Information

Another difficult issue related to the UFS file system and the VM system is dealing with the control information that the *vnode* driver uses to manage the logical file. For the UFS implementation, the control information consists of the *inodes,* indirect blocks, cylinder groups, and super blocks. The control information is not part of the logical file and thus the control information still needs to be named by the block device offsets, not the logical file offsets. To provide the greatest flexibility we decided to retain the old buffer cache code with certain modifications for optional use by file system implementations. The biggest driving force behind this is that we did not want to rely on the system page size being smaller than or equal to the size of control information boundaries for all file system implementations. Other reasons for maintaining parts of the old buffer cache code included some compatibility issues for customer written drivers and file systems. In current versions of SunOS, what's left of the old buffer cache is used strictly for UFS control buffers. We did improve the old buffer code so that buffers are allocated and freed dynamically. If no file system types choose to use the old buffer cache code (e.g., a diskless system), then no physical memory will be allocated to this pool. When the buffer cache is being used (e.g., for control information for UFS file systems), memory allocated to the buffer pool will be freed when demand for these system resources decreases.

9. System Changes

With the conversion to the new VM system, many closely related parts of the SunOS kernel required change as well. For the most part time constraints persuaded us to retain the old algorithms and policies.

9.1. Paging

The use of the global clock replacement algorithm implemented in 4.2BSD and extended in 4.3BSD is retained under the new VM system. The "clock hands" now sweep over *page* structures, calling *hat_pagesync()* on each eligible *page* to sync back the reference and modified bits from all the hat translations to that page. If a dirty page needs to be written back, the page daemon uses VOP_PUTPAGE() to write back the dirty page.

9.2. Swapping

We retained the basic notion of "swapping" a process. Under the new VM system there is much more sharing going on than was present in 4.2BSD where the only sharing was done explicitly via the *text* table. Now a process's address space may have several shared mappings, making it more difficult to understand the memory demands for an address space. This fact is made more obvious with the use of shared libraries [9, 10].

The address space provides an address space swap out operation *as_swapout()* which the SunOS kernel uses when swapping out a process. This procedure handles writing back dirty pages that the *as* maps and that no longer have any MMU translations after all the resources for the *as* being swapped are freed. The *as_swapout()* operation returns the number of bytes actually freed by the swap out operation. The swapper saves this value as a working set estimate[12], using it later to determine when enough memory has become available to swap the process back in. Also written back on a process swap out operation is the process's user area, which is set up to look like anonymous memory pages.

The *as* and segment structures used to describe the machine independent mappings of the address space for the process are currently not swapped out with the process since we don't yet have the needed support in

[11] The *inode* is the file system private *vnode* information used by the UFS file system [2].

[12] Unfortunately, a poor one; this is an opportunity for future improvement.

the kernel dynamic memory allocator. This differs from the 4.2BSD VM implementation where the page tables used to describe the address space are written back as part of the swap out operation.

9.3. System Calls

We rewrote many traditional UNIX system calls to manipulate the process's user address space. These calls include *fork*, *exec*, *brk*, and *ptrace*. For example, the *fork* system call uses an address space duplication operation. An *exec* system call destroys the old address space. For a demand paged executable it then creates a new address space using mappings to the executable file. For further discussion on how these system calls were implemented as address space operations see [1].

Memory management related system calls based on the original 4.2BSD specification [11] that were implemented include *mmap*, *munmap*, *mprotect*, *madvise*, and *mincore*. In addition, the *msync* system call was defined and implemented. For further discussion on these system calls see [1].

9.4. User Area

The UNIX user area is typically used to hold the process's supervisor state stack and other per-process information that is needed only when the process is in core. Currently the user area for a SunOS UNIX process is still at a fixed virtual address as is done with most traditional UNIX systems. However, the user area is specially managed so context switching can be done as quickly as possible using a fixed virtual address. There are several reasons why we want to convert to a scheme where the user areas are at different virtual addresses in the kernel's address space. Among them are faster context switching[13], better support for multi-threaded address spaces, and a more uniform treatment of kernel memory. In particular, we are moving toward a **seg_u** driver that can be used to manage a chunk of kernel virtual memory for use as u-areas.

10. Performance

A project goal for the new VM work was to provide more functionality without degrading performance. However, we have found that certain benchmarks show substantial performance improvements because of the much larger cache available for I/O operations. There is still much that can be done to the system as a whole by taking advantage of the new facilities.

Table 2 shows some benchmarks that highlight the effects of the new VM system and dynamically linked shared libraries [9, 10] over SunOS Release 3.2. *Dynamic* linking refers to delaying the final link edit process until run time. The traditional UNIX model is based on *static* linking in which executable programs are completely bound to all their libraries routines at program link time using *ld*(1).

Kernel Tested	SunOS 3.2	Pre-release New VM	Pre-release New VM
Binaries Executed	3.2	3.2	Dynamically Linked
Tests Performed	**Time (secs)**	**Time (secs)**	**Time (secs)**
exec 112k program 100 times	7.3	3.3	10.7
fork 112k program 200 times	8.8	4.4	7.7
Recursive stat of 125 directories	4.9	1.4	1.3
Page out 1 Mb to swap space	2.0	2.0	0.8
Page in 1 Mb from swap space	4.6	3.8	3.5
Demand page in 1 Mb executable	1.7	0.9	0.8
Sequentially read 1 Mb file (1st time)	1.6	1.5	1.5
Sequentially read 1 Mb file (2nd time)	1.6	0.4	0.4
Random read of 1 Mb file	5.7	0.7	0.8
Create and delete 100 tmp files 6.3	4.7	4.7	

Table 2
System Benchmark Tests on a Sun-3/160
with 4 Megabytes of Memory and an Eagle Disk

[13] This is especially true with a virtual address cache and a fixed user area virtual address, since the old user area must be flushed before the mapping to the new user area at the same virtual address can be established.

Running a new VM kernel with same 3.2 binaries clearly shows that the new VM system and its associated file system changes has a positive performance impact. The effect of the larger system caching effects can be seen in the read times.

One way that the system uses the new VM architecture is a dynamically linked shared library facility. The *fork* and *exec* benchmarks show that the flexibility provided by this facility is not free. However, the benefits of the VM architecture that provides copy-on-write facilities more than compensate for the cost of duplicating mappings to shared libraries in the *fork* benchmark. The *exec* benchmark is the only test that showed performance degradation from dynamically linked executables over statically linked executables run with a SunOS Release 3.2 kernel. These numbers show that the startup cost associated with dynamically linking at run time is currently about 74 milliseconds. These results are preliminary and more work will be undertaken to streamline startup costs for dynamically linked executables. We feel that the added functionality provided by the dynamic binding facilities more than offsets the performance loss for programs dominated by start up costs.

11. Future Work

The largest remaining task is to incorporate better resource control policies so that the system can make more intelligent decisions based on increased system knowledge. We plan to investigate new management policies for page replacement and for better integration of memory and processor scheduling. We believe that the VM abstractions already devised will provide the hooks needed to do this. SunOS kernel ports to different uniprocessor and multiprocessor machine bases will provide further understanding of the usability of the abstractions and our success in isolating machine dependencies.

Other future work involves taking advantage of the foundation established with the new VM architecture—both at the kernel and user level. Specialized segment drivers can be used at the kernel level to more elegantly support various unique hardware devices and to support new functionality such as mappings to other address spaces. Shared libraries are an example of the usefulness of mapped files at the user level. We expect to find the features of the new VM system used in various new facilities yet to be imagined. As new uses for the VM system are better understood, we can refine and complete the interfaces that have not yet been fully defined.

12. Conclusions

From our experience in implementing the new VM system, we draw the following conclusions.

- **Object oriented programming works.** The design of the new VM system was done using object-oriented techniques. This provided a coherent framework in which we could view the system.
- **The balance of responsibility is important.** When partitioning a problem amongst different abstractions, it is critical that the system be structured so that each abstraction has the right level of responsibility. When an abstraction gets control at the right time it has the opportunity to recognize and act on events that make sense for that abstraction.
- **The layering in the new VM system is effective.** For example, the *hat* layer provides all the machine dependent MMU translation control and has been found to be easily ported to new hardware architectures. The use of segment drivers has proven to make the system more extensible.
- **Performance did not suffer.** Although the new VM system provides considerably more functionality, it did so without any performance loss. Performance often improved because the new VM system better uses memory resources as a cache. By carefully designing the abstractions with optimizations for critical functions, we reduced the cost sometimes associated with object-oriented techniques to provide clean abstractions that are still efficient.

13. Acknowledgements

I would like to thank Rob Gingell, Dave Labuda, Bill Shannon, and especially Glenn Skinner, for reviewing this paper and helping to make it presentable and for their work with the new VM system. And most of all I would like to give a big thank you to my understanding wife Laurel, who continued to tolerate me during the countless extra hours I put into this project.

14. References

[1] Gingell, R. A., J. P. Moran, W. A. Shannon, ''Virtual Memory Architecture in SunOS'', *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[2] Kleiman, S. R., ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX'', *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[3] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley Publishing Company, 1986.

[4] Rashid, R., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, J. Chew, ''Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures'', *Operating Systems Review*, Volume 21, No. 4, October, 1987.

[5] Kepecs, J. H., ''Lightweight Processes for UNIX Implementation and Applications'', *Summer Conference Proceedings, Portland 1985*, USENIX Association, 1985.

[6] Sun Microsystems Inc., *Sun-3 Architecture: A Sun Technical Report*, 1985.

[7] Cheng, R., ''Virtual Address Cache in UNIX'', *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[8] McKusick, M. K., W. N. Joy, S. J. Leffler, R. S. Fabry, ''A Fast File System for UNIX'', *Transactions on Computer Systems*, Volume 2, No. 3, August, 1984.

[9] Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, ''Shared Libraries in SunOS'', *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[10] Gingell, R. A., ''Evolution of the SunOS Programming Environment'', *Spring Conference Proceedings, London 1988*, EUUG, 1988.

[11] Joy, W. N., R. S. Fabry, S. J. Leffler, M. K. McKusick, *4.2BSD System Manual*, Computer Systems Research Group, Computer Science Division, University of California, Berkeley, 1983.

System V Release 3, Diskless Workstations and NFS

Robert Cranmer-Gordon, Bill Fraser-Campbell, Mike Kelly and Peter Tyrrell
rob@inset.co.uk, bill@inset.co.uk, wot@inset.co.uk, petet@inset.co.uk

The Instruction Set

ABSTRACT

Diskless UNIX workstations are becoming a fashionable way of providing users with high levels of facilities and performance at low cost. To date, most implementations of UNIX for diskless computers have been based on 4.2BSD. This paper describes some major modifications made to Motorola System V/68 to produce a version of System V Release 3 capable of supporting diskless machines.

To make diskless operation possible using Sun's Network File System (NFS over Ethernet, the File System Switch feature of System V Release 3 has been replaced with the Sun Virtual File System (VFS) switching arrangement. However, the Release 3 STREAMS architecture has been retained as a framework for Internet protocol software to permit a (comparatively) easy switch to OSI protocols in the future. The BOOTP (RFC 951) and TFTP protocols are used for bootstrapping diskless machines.

The paper presents details of the method used to marry NFS and STREAMS, performance enhancements to the Sun distributed record locking and experiences with BOOTP. It also lists those awkward places where the requirements of NFS and the System V Interface Definition (SVID) conflict.

1. Introduction

Until recently, most diskless workstations have been aimed at a specific market sector, that of computer aided design and engineering. The cost of high resolution graphics hardware, Ethernet connectivity and ample cpu performance has mitigated against other, more general applications. Now that the cost of the hardware has come down to more affordable levels commercial uses of such workstations are practical. In this paper we describe the implementation of UNIX software on a diskless graphics workstation designed to provide distributed computing in a commercial environment.

The prototype hardware is fairly simple. The processor is a Motorola 68020 with 68851 paged memory management unit and 4 Megabytes of RAM. 2 serial channels, a SCSI bus interface, 2.5 Megapixels of video memory and an Ethernet interface are provided.

1.1. Software

The commercial application area and the processor type made Motorola System V/68 Release 3 the obvious choice for the operating system base (4.*x* BSD addicts may choose to differ). Several additional constraints influenced the port:

i. Compatibility with the SVID[1].

ii. The availability of STREAMS to support a variety of protocols in a consistent way.

iii. Support for the Network File System (NFS)[2] so that a variety of different machines could be used as file servers.

System V/68 meets the first two of these, but extensions were necessary in three areas:

i. V/68 supports Remote File Sharing (RFS) but not NFS.

ii. There is no communications protocol software included in the V/68 distribution. NFS normally runs over UDP/IP.

iii. There is no support for diskless operation. This implies work in the areas of bootstrapping, starting up communications, locating a root file system, and making access to pipes and devices possible.

2. The Vnode Kernel

NFS in a UNIX kernel is normally supported as a new file system type underneath a file system switching arrangement. In the Sun operating system this switch is called the Virtual File System (VFS). System V Release 3 provides a similar layer, called the File System Switch (FSS). The VFS and FSS interfaces are quite different, the FSS being based on System V Release 2 internal interfaces (*iget()*, *readi()* etc), and the VFS providing more abstract operations (*vn_lookup()*, *vn_rdwr()* etc) [3]. Two options were open to us: to rewrite the NFS file system to suit the FSS interface or to rewrite the System V Release 3 kernel to use the VFS. We chose to use the VFS for the following reasons:

i. The FSS in its present form is not a stable interface. AT&T describe it as "unpublished", and we believe that it will change in the future.

ii. Fitting the VFS into a System V kernel, albeit Release 2, is known to be straightforward [4].

iii. The FSS does not support symbolic links. These are almost essential for diskless operation as they make it practical for diskless stations to share the unchanging parts of the root file system.

iv. The NFS is much more difficult to debug than a local file system because a network is involved. One of our goals in this project was to minimise the changes necessary to the existing NFS software.

v. Replacing the FSS with the VFS does not interfere greatly with the SVID compatibility of the finished system. In particular, it does not preclude support for STREAMS. The only difficult part of STREAMS to support in a VFS kernel is clone devices, where "new" devices have to be invented on demand [5].

vi. The VFS provides diskless workstations with the specfs, which supports pipe buffering without reference to disk, and also supports the use of remotely stored special file nodes which refer to local devices.

Against this, changing to the VFS architecture meant losing support for Remote File Sharing (RFS) in the prototype system.

3. NFS and STREAMS

NFS operations on a client machine use a Remote Procedure Call (RPC) protocol to pass requests to the file server. RPC requests are encoded by an eXternal Data Representation (XDR) protocol to resolve byte-ordering problems between machines. The output of this XDR encoding is placed in message buffers ready to be given to UDP/IP for transmission. The numbers in the table below show a very approximate correspondence to OSI layers.

NFS	(7)
RPC/XDR	(5/6)
UDP	(4)
IP	(3)

Table 1: NFS protocol stack

Since NFS operations are entirely managed within the kernel, no user memory is involved in this processing. User read/write data is copied in or out of the kernel address space by the file system independent code above the VFS. The two areas where modifications were necessary to the NFS code were the management of the in-kernel message buffers, and the interface between the RPC code and UDP/IP.

3.1. Message Buffering

In the Sun NFS implementation, XDR deposits the data to be transmitted into buffers called mbufs. This is the normal communications buffering scheme used by 4.*x* BSD. Mbufs are not provided in System V. Instead, System V Release 3 uses a STREAMS buffering mechanism. We chose a simple method to resolve the differences between mbufs and STREAMS buffering.

When an RPC transaction is started, a contiguous buffer large enough to hold the maximum size of UDP packet is allocated from a kernel heap for both the sent message and the received reply. The XDR code assembles the message in the transmit buffer. This neatly sidesteps any complications of message fragmentation, and simplifies the XDR code at the same time, since it no longer has to understand mbuf structures. The complete message is then copied into a single STREAMS buffer and passed to UDP/IP. When the STREAMS buffer is discarded at the end of transmission, the copy in the heap remains available for retransmission if necessary.

3.2. Interface to UDP/IP

The Sun NFS code uses sockets to pass data from RPC to UDP protocol code. Being inside the kernel, internal interfaces are used:

socreate()	create a socket
sobind()	bind a port number
ku_sendto_mbuf()	send a UDP message
ku_recvfrom()	receive a UDP message
soclose()	destroy a socket

The STREAMS mechanism lacks even the concept of a socket, so none of these interfaces exist although analogues are provided by the Transport Layer Interface [6].

Two alternative methods of providing a socket interface were considered:

i. Devise a new STREAMS module to provide the missing 5 functions.

ii. Choose a set of TCP/IP STREAMS modules which already include a socket emulation, and explore the internal interfaces they provide.

The availability of a set of STREAMS TCP/IP modules (from Lachman Associates Inc. of Chicago) which include a full emulation of Berkeley sockets decided us on alternative (ii). This suite of modules provides primitives for socket creation, binding and destruction suitable for use from inside the kernel. It also provides for the transmission and reception of data, but quite reasonably expects the data to be in user program address space.

Fortunately this limitation turned out to be a disguised opportunity for performance improvements. Modified versions of *sendto()* and *recvfrom()* were prepared which expected data in the kernel virtual address space, and at the same time left out code for handling out-of-band data or protocols other than UDP. User level access to UDP/IP is not affected.

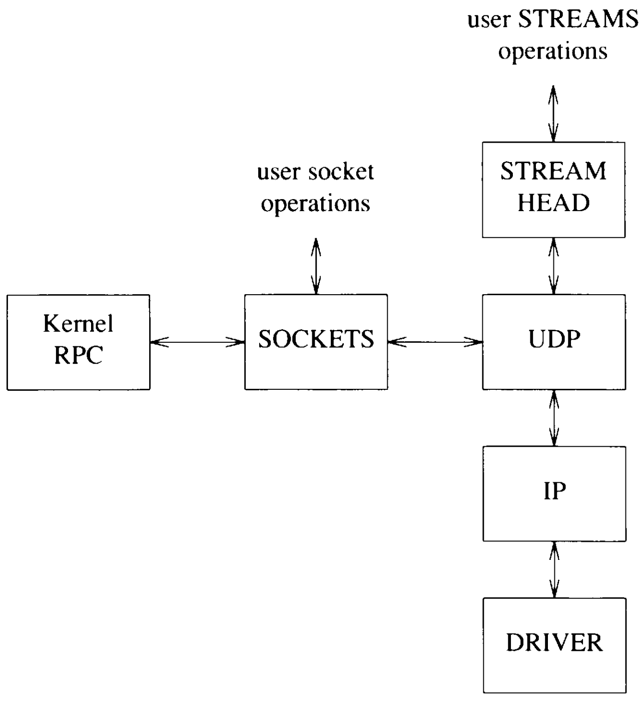


Table 2: Kernel RPC and STREAMS

Thus our kernel RPC uses side-entries to the socket emulation package, which in turn uses side-entries to the STREAMS protocol stack.

4. Using the RFC951 Bootstrap Protocol

The diskless workstation, faced with the problem of finding a kernel to load and run, and a root filesystem to use, needs help: it has somehow to request that a server transfer the relevant file and information. The BOOTP protocol, described in RFC 951 [7] allows such a workstation, armed only with the knowledge of its own hardware address, to discover its own IP address, that of a server, and the name of a file to be loaded and executed. This information can then be used in a second phase to actually transfer the file, typically using the TFTP protocol (RFC 783 [8]). The simplicity of both these protocols allows them to reside in PROM, such that the workstation may boot without user intervention.

4.1. BOOTP

The BOOTP protocol involves a single UDP/IP datagram packet exchange, the same packet format being used in both directions. Both transmissions are sent to the broadcast IP address, and two reserved UDP port numbers are used: `` BOOTP server'' (67) and `` BOOTP client'' (68). The essential elements are:

i. an ``opcode'' field, specifying either BOOTREQUEST or BOOTREPLY,

ii. the client's hardware address field, used in its request,

iii. two fields for ``your and my IP addresses'', used in the server's reply for the client's and its own IP addresses,

iv. a field for the filename to be loaded, which may optionally be set by the client to request a particular file, but which (for security reasons) the server may choose to ignore and reply with a default filename.

The protocol also allows for optional vendor-specific information, used in our implementation for passing into the executed kernel the filehandle for its root filesystem, along with the client and server IP addresses. Implementing BOOTP so that this information is treated opaquely allows changes to be made to the server's boot daemon and the kernel without the need to change the boot software in PROM.

Since the workstation probably only knows its hardware (Ethernet) address, its initial boot request has to be sent to the broadcast IP address. The server must then look this up in its boot database, to find the client's IP address to send as ``your address'' in its reply. If the received hardware address is not found in the database, the request is simply ignored, since the server cannot do anything useful with it.

4.2. Implementation

Variations from BOOTP

An optional part of the protocol is the handling of booting from a server on a different network through a gateway. Two fields in the request and reply packets can be used to specify the address of a gateway, and the number of ``hops'' a packet can make. Given the likely topology of the network in which the workstations will be used, this was not thought to be worth implementing.

RFC 951 allows for the case of having a reserve server in case the normal server fails. We did not consider this, since only the normal server has the diskless workstation's root filesystem.

RFC 951 does not handle the idea of boot failure. If the server cannot boot the client it simply ignores the request. The client will timeout and retransmit forever. An addition was made to the protocol to handle internal server errors. The server returns the opcode BOOTFAILURE and the client can then display a message and terminate the boot attempt. The user is then aware that the error lies with the server, and not the client or network.

Boot Options

As described above, the aim was for a workstation which would boot automatically without user intervention. An *ebo* (Ethernet boot) command was added to the machine's existing monitor to allow for a manual boot.

ebo [*server-name*:][*file*] [*client-IP-address*]

server-name
: may be either the actual host name or its IP address. It is passed in the boot request's *sname* field, which is otherwise normally left empty. The server boot daemon ignores any requests that have *sname* set to something other than its own name.

file is the name of the file being requested to be loaded and executed. For security reasons, the boot daemon may choose to ignore this filename and reply with a default for that client, as specified in the boot database (see below).

client-IP-address
allows a machine to ask to be booted as a different machine, which is is useful for testing purposes. It also allows a server whose own root file system has been corrupted to boot as a diskless machine.

Boot Database

The boot daemon *(bootd)* on the server awaits the receipt of UDP packets on the BOOTP server's reserved port. It then does a look-up in the boot database on the Ethernet address from the request.

An example of the format used for this database is shown below. Since information such as the server name and default file to boot will be common between different clients, they can be specified once in a DEFAULT section. Client-specific information, such as the Ethernet address, can then be listed individually.

```
# sample bootd database

1DEFAULT
        internet = usehosts
        server = dyadic
        default = unix

ritchie
        ether = 12.0.fa.ce.9f.d3
        rootfs = /usr/client/ritchie
        default = unix.new

kernighan
        ether = 12.0.fa.ce.b1.fe
        rootfs = /usr/client/kernighan
```

The "internet = usehosts" entry specifies that the */etc/hosts* file is to be used as the default way of mapping machine-name to IP address.

If the boot file is not an absolute pathname, then it is taken to be relative to the root directory *(rootfs)* for that client. For example, the default file to boot the client *ritchie* is */usr/client/ritchie/unix.new*.

Modifications to TFTP

Two modifications were made to TFTP to shorten the time spent loading the kernel:

i. the server's TFTP daemon was modified to use the size of the UDP data in its first received packet as the packet size for all its subsequent transmissions.
The TFTP protocol specifies the packet size to be 512 data bytes, but if the client now sends its file transfer request in a 1400 data byte packet, the server can reply with the same sized packet. During development, the data size used by the client could be specified by an option to the *ebo* command, and empirical evidence showed that the larger packet size resulted in a shorter transfer time.

ii. the STOP opcode was added to the TFTP protocol so that the client could cleanly cease the transfer once it had received the COFF header, text and data sections, but avoiding the transfer of the symbol table. For our kernel, this represents a saving of 16%.

Vendor Information

BOOTP allows for an optional 64 bytes of vendor-specific information at the end of the boot reply packet. By using the structure defined below, our implementation of the boot code treats the information opaquely. It only has to "know" about the header, from which it obtains the size of the whole structure.

```
/*
 * Common diskless boot information structure header.
 */
struct dbihdr {
        long    dbih_magic;      /* Magic no. */
        long    dbih_length;     /* Size of this struct */
};

struct dbinfo {
        struct dbihdr           dbi_header;

        char                    dbi_servname[DBISNLEN];
        struct sockaddr_in      dbi_servip;
        struct sockaddr_in      dbi_clntip;
        fhandle_t               dbi_rootfs;
};
```

When received in the boot reply, the structure is copied to some safe location before the kernel is loaded. This location, the magic number and length are passed into the executed kernel via certain registers. The kernel can then determine whether it is diskless or not, and if it is, obtain the necessary information from this structure.

5. Diskless System V Release 3

For this project diskless operation must be performed using an NFS remote root filesystem. This introduces a number of problems, either as a result of deficiencies in NFS, or introduced by the System V environment.

5.1. Getting Started

In the normal NFS environment, some user process involvement is required before an NFS filesystem can be mounted:

i. Network initialisation
   Before any activity involving remote machines may take place the client must be able to access the network. In our environment the network initialisation is performed by several user programs; *slink*, *ldsocket* and *ifconfig*. These programs access data in files such as */etc/netcf* to determine how the streams and sockets should be configured, and then devices to do the configuration.

ii. Mounting an NFS filesystem
   Before the mount request is passed to the kernel, the client mount program contacts the server mount daemon to obtain the file handle for the remote filesystem root directory. The file handle and the server name and internet address are passed to the kernel by the mount system call.

In the diskless environment we are unable to do the above as the root filesystem has not yet been mounted and so are unable to use any configuration files or programs. Here we have the classic chicken and egg situation

5.2. Managing without Local Disks

The standard UNIX kernel assumes the presence of local disk devices. Problems occur in the following area:

i. Pipes and clones
   Normally the inodes for pipe files and cloned devices, although invisible to the user, are allocated from the root filesystem. This source of inodes has now disappeared.

ii. Swapping
   The kernel expects a local disk on which it can swap. This has disappeared.

5.3. Other Diskless Problems

This problem has disappeared with the release of NFS 3.2 but is included here for completeness.

NFS does not support remote device access, although remote device files may appear to be opened. If your root filesystem is remote, then how do you talk to local devices such as the console?

5.4. Network Initialisation

For us the stream and socket configuration will be static, so we are able to hard code into the kernel the necessary initialisation. Data such as the client internet address is passed into the kernel by the bootstrap, so it is available when the kernel version of *ifconfig* takes place.

The kernel initialisation mimics the user version, except that it is able to call functions directly. Close inspection revealed that most functions can be invoked from the kernel, or that appropriate functions could be derived where changes only relate to the choice of kernel or user address space for arguments.

The root mounting takes place in *iinit()*, so the stream and socket initialisation must have occurred by this time. In fact we do it as part of *nfs_mountroot()* itself. This also means that a non-diskless system will initialise the network in the original manner.

From a user process, stream and socket initialisation consists of opening the relevant device(s), and pushing or linking the modules. The kernel version needs to do the same so we require a process context in which to open files. Luckily, by this time we have the first u-area set up, so we are able to call *falloc()*. We are able to fake up the driver and protocol vnodes.

An important point to remember is that the *slink* program keeps the initialised streams open, otherwise the streams will be dismantled by the close. In the diskless case we do not have a user process to do the job so it must be achieved in a slightly devious manner. After initialisation cleanup code clears out the per-process file table without closing the files. The result is that the files have file table entries with non-zero use counts causing the files to remain open forever. The per-process file table must also be cleared out to stop children from inheriting the streams.

5.5. Mounting the Root Filesystem

The problem of obtaining the root directory file handle, the server name and server internet address has been solved by passing the task to the bootstrap, which passes it into the kernel when run. With this information mounting the NFS root becomes simple.

5.6. Pipes and Clones

Pipe vnodes are now created by a call to *fakevp()* which can be used to create a fake vnode of any type. This function is also used when vnodes are required by cloning.

5.7. Swapping

We have chosen to allow diskless machines to swap into files held on the server. The file */dev/swap* is created in the client's root filesystem. When the client boots, it can use VOP_GETATTR() to find the extent of the file. The client never attempts to grow the swap file, so it will not take all of the servers disk space. The system administrator sets the swap size by creating the swap file. Also note the swap file may be created as a sparse file, in which case lots of swap may be allocated as long as none of the clients try to use it.

The vnode swap code uses VOP_STRATEGY() to perform I/O so it will work with a remote file.

5.8. Device Access

If the root filesystem is remote, then some method is required to force remote device nodes to refer to local devices. This problem may be overcome, and has been in NFS release 3.2, by the use of the specfs. When the name lookup finds a device node, it uses to create an snode, in which the VOP_XX() entries treat the device as local. The specfs is used for both local and remote device nodes.

6. Distributed Record Locking

As part of the work on converging the 4.*x* BSD and System V operating systems, Sun Microsystems has developed a SVID compatible file and record locking mechanism which works with both local and NFS files. This code has been released along with the upgrade to NFS release 3.2.1.

Lock requests are handled using a separate protocol which exists along side NFS, the protocol only allowing for advisory locking.

The Sun lock code seems the obvious place to start when implementing remote locking with NFS.

6.1. Overview of the Sun Locking Mechanism

The whole of the lock administration takes place at process level. The kernel forwards lock requests, by RPC, to a local user process called the lock manager.

Lock information for a file resides on the same machine as the file itself. The lock database for a machine is held by that machines lock manager.

When a lock request is made, the local (client) lock manager determines if the lock refers to a local file. If so the lock manager can grant or deny the request itself. But in the case of a request for a remote lock, the client lock manager must contact the remote (server) lock manager, which holds the lock database for all files residing on the remote machine. The server lock manager will then grant or deny the lock.

Remote file locking is stateful. If either the client or server crash the lock information held by the other becomes incorrect. In the case of a server crash, the client loses its locks, but can still access the file as NFS recovers. If the client crashes, the server will still retain the locks on behalf of the client. The statelessness of NFS is not compromised by network locking, as the two exist separately but alongside.

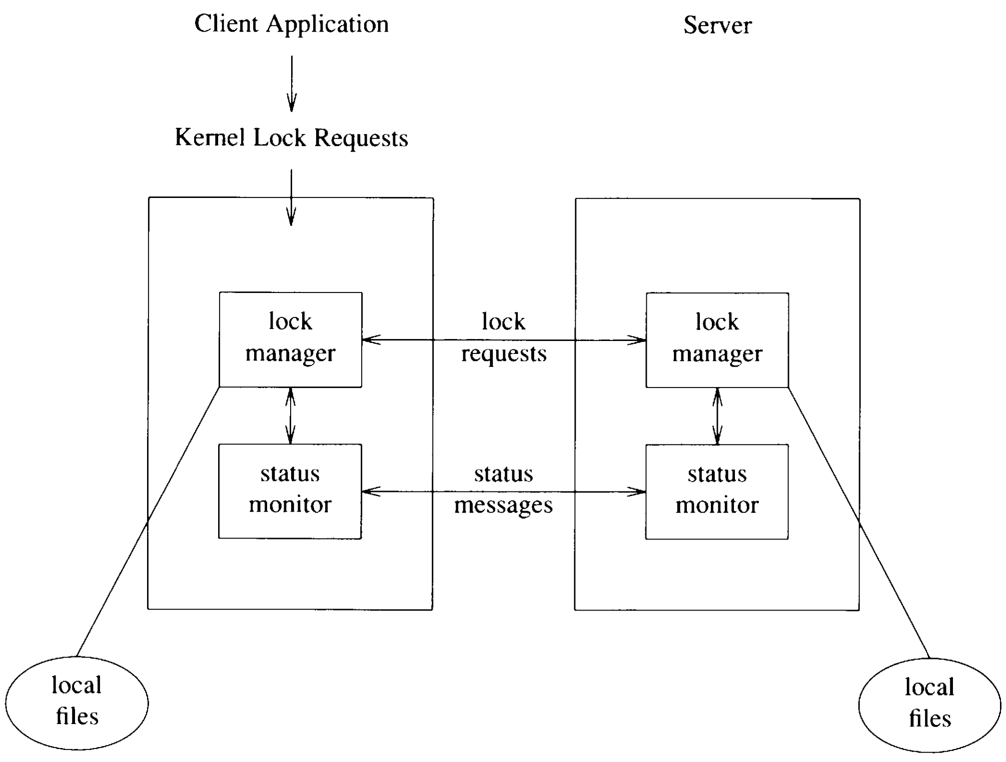


Figure 3: Lock Manager Architecture

Crash Recovery

Crash recovery occurs when the client or server comes back up. It is achieved using two mechanisms:

i. Duplication of lock state information

   Both the client and server lock managers hold lock information. The server knows which locks it has allocated to which client, and the client knows which locks are held or requested on which server.

ii. Status monitor

   The status monitor is a process whose task is to monitor the state of other machines. It informs the local lock manager when crash recovery is required.

Recovery from a client crash is not too difficult. The client will have lost all state and need to restart from scratch again, so the server just removes all locks held by that client.

Recovery from a server crash is more complicated as its clients require the previous state to be regained. This is possible because each client knows the locks it was holding. The server lock manager enters a grace period in which it allows clients to replace old locks. Each client lock manager should attempt to recover its locks during the grace period. After this time all locks are considered new.

Complete recovery is not guaranteed, clients may lose locks. When this situation arises the client lock manager uses the *SIGLOST* signal to inform the client application of the loss. It is up to the application to take remedial action.

6.2. Implementation Alternatives

i. Port the Sun lock manager as is, replacing the System V Release 3 locking.
   The porting effort here should be lower, but there are disadvantages: every lock request has the overhead of an RPC call to the local lock manager, the lock manager must be running before any locks may be attempted, and mandatory locking is not practical.

ii. Retain System V Release 3 locking for local files, and use the Sun locking for remote files.
   The advantages of this solution are: no overhead due to the lock manager on local files, mandatory locks are available on local files, and the lock manager is only required for remote locking. The disadvantage is that a server lock manager must be able to verify the validity of a remote lock when it knows nothing about locks placed by local processes.

6.3. Our Solution

We decided to retain System V Release 3 locking and modify the Sun network lock management to fit this.

The vnoding of the kernel provided local file locking which does not need modification to work with the network locking.

The real change is to the server side of the lock manager, which must now communicate with the kernel lock database to coordinate local and remote lock placement.

The lock manager attempts to place the lock using a new system call *lockctl()*, which provides locking facilities similar to *fcntl()*. The lock arbitration takes place in the server kernel using the System V Release 3 locking. The lock manager accepts the lock if the kernel does. This means that the server lock manager database is now really only used for client crash recovery.

The lock manager cannot use *fcntl()* because it does not have the filename with which it can open the file, only the file handle, and also cannot be expected to keep open all files on which it has placed locks. The *lockctl()* gets around the problem by accepting a file handle, and by modifying the inode count so the file remains ''open'' whilst any remote locks are placed.

Although *lockctl()* supports both blocking and non-blocking lock requests the lock manager must never use a blocking request, as a blocked request will delay all further client locking until the first lock is placed.

The Sun implementation for the client kernel-to-lock-manager RPC request makes use of the portmapper to find the lock manager port number. This was considered to involve too much overhead and would also require the addition of portmapper code in the kernel. Another solution is to use a ''well known'' port number or to have the lock manager register its port number by system call.

System V Release 3 does not have the signal *SIGLOST* for use when client locks are lost. For this feature to be used, we either add the signal, or map it onto an existing signal. Part of our project testing consists of the complete rebuild of the system, so adding the new signal does not cause any problem.

7. SVID Conformance

At the end of the prototype port, the system conforms to the System V Interface Definition as well as Motorola System V/68 does, with the following restrictions, enhancements and extensions:

7.1. Restrictions

i. The *mknod()*, *link()* and *unlink()* system calls cannot be used to create and remove directories. The NFS protocol does not support non-atomic operations, so the System V Release 3 *mkdir()* and *rmdir()* system calls must be used.

ii. Mandatory record locking cannot be enforced on remote files.

iii. Fifo files (named pipes) cannot be used to communicate between processes on different machines, although they work perfectly within a single client. This is because there is no provision in the NFS protocol for sending SIGPIPE signals across the network. It is also undesirable to have server processes tied up waiting for data to arrive on a pipe.

iii. A file server cannot distinguish between read accesses from a client made as part of a *read()* and read accesses for demand paging. The distinction between read and execute access permissions on files is therefore lost.

7.2. Enhancements

System calls now all produce SVID conformant error numbers.

7.3. Extensions

Eight new system calls are added as part of the NFS work:

i. *rename()* is a new system call which can be used to rename files atomically. *link()* and *unlink()* can still be used, but are less secure in the event of system failure.

ii. *domainname()* allows a Yellow Pages domain name to be written and retrieved on demand.

iii. *vmount()* accepts a filesystem type argument and so can be used to mount any type of filesystem.

iv. *vumount()* can unmount any type of filesystem.

v. *async_daemon()* causes the calling process to become an asynchronous I/O handler. It is only used by the *biod* program.

vi. *nfs_getfh()* allows the *mountd* daemon program to generate an NFS file handle for a named file.

vii. *nfs_svc()* turns the calling process into an NFS file server process. It is only used by the *nfsd* program.

viii. *lockctl()* allows a server lock manager to place a lock on a local file on behalf of a remote client process.

In addition, the Berkeley *mtab* and *fstab* file formats are used to store details of mounted file systems.

8. Acknowledgements

Thanks are due to a number of people for their help in this project:

Lynwood Scientific Developments Ltd initiated the project and supplied the hardware and the majority of the development team. The software team was Martin Tann, Kevin Law, Dave Gordon, Dave Griffiths, Peter Mills-Baker, Gerard O'Driscoll, Noel Poore, Nick Stoughton, Andy Guilbert and the authors. Dick Young and Mark Collingwood designed and supported the hardware.

9. References

1. *The System V Interface Definition*, Issue 2, AT&T 1986.
2. *Design and Implementation of the Sun Network File System*, Sandberg et al, Sun Microsystems Inc., 1985.
3. Vnodes: An Architecture for Multiple File System Types in Sun UNIX, S.R. Kleiman, USENIX Conference Proceedings Summer 1986.
4. ``An Implementation of NFS under System V.2'', Bill Fraser-Campbell and Mordecai B. Rosen, *EUUG Conference Proceedings*, Spring 1986.
5. *UNIX System V STREAMS Programmer's Guide*, AT&T.
6. *UNIX System V Network Programmer's Guide*, AT&T.
7. *Bootstrap Protocol (BOOTP)*, RFC 951, Bill Croft and John Gilmore, NIC, September, 1985.
8. *The TFTP Protocol*, RFC 783, K. R. Sollins and Noel Chappa, NIC, June, 1981.

The Andrew Toolkit – An Overview

Andrew Palay
et al.

Carnegie Mellon University

ABSTRACT

The Andrew Toolkit is an object-orientated system designed to provide a foundation on which a large number of diverse user-interface applications can be developed. With the Toolkit, the programmer can piece together components such as text, buttons,and scroll bars to form more complex components. It also allows for the embedding of components inside other components, such as a table inside of text or a drawing inside of a table, Some of the components included in the Toolkit are multi-font text, tables. spreadsheets, drawings, equation, rasters, and simple animations. Using these components we have built a multi-media editor, a mail-system, and a help system. The Toolkit is written in C, using a simple preprocessor to provide an object-oriented enviroment. That enviroment also provides for the dynamic loading and linking of code. The dynamic facility provides a powerful extension mechanism and allows the set of components used by an application to be virtually unlimited. The Andrew Toolkit has been designed to be window-system independent. It currently runs on two window systems, including X.11, and can be ported easily to others.

(This paper was not submitted in time for publication.)

European UNIX systems User Group
Owles Hall, Buntingford
Hertfordshire, SG9 9PL
England
tel: +44 763 73039
fax: +44 763 73255
email: euug@inset.uucp
or ...!mcvax!ukc!inset!euug

ISBN 0-9513181-0-1

PRINTED BY THE HOLYWELL PRESS LTD, ALFRED STREET, OXFORD