KimCM

E U U G

European UNIX™ systems User Group

SPRING CONFERENCE PROCEEDINGS

Florence, Italy
April 21-24 1986

Further copies of the proceedings may be obtained from:

EUUG Secretariat
Owles Hall
Buntingford
Herts
SG9 9PL
United Kingdom

EDITORIAL

This set of proceedings is organised as follows:

- The programme at a glance (thank you TBL), including who speaks when;
- List of speakers and their papers in alphabetic order;
- The set of abstracts submitted, on which the selection of papers were based;
- The actual papers submitted, also in alphabetic order.

It was unfortunately not possible to await the arrival of all papers before the proceedings were sent to the publisher. Therefore this gave us an excuse to utilise the wonderful PIC that appears against every page for which we did not manage to obtain a paper.

Some acknowledgements are in order: firstly to the authors for submitting most of the papers in a reasonable form and usually only a short time after the absolute final deadline; secondly to The Instruction Set Limited for the use of their facilities to typeset the majority of the artwork for the proceedings; and finally to Mike Kelly, of The Instruction Set who assisted greatly with the actual production of this material.

Finally it is appropriate to thank the many authors who submitted very good papers, but were not selected for this programme. It is always very difficult to perform the selection, and it is becoming more so as the quality of papers increases. In many cases, the reason for rejection was not that the paper was in any way below standard; rather that the topics addressed were not appropriate for the material planned for this conference. Such authors should, if appropriate, submit their papers again for the next EUUG conference in Manchester.

Nigel Martin
The Instruction Set Limited

TECHNICAL PROGRAMME TIMETABLE

	Tuesday 22nd	Wednesday 23rd	Thursday 24th
09:20	i2u (Italy) News	FUUG (Finland) News	NUUG (Norway) News
09:30	Massimo Bologna The Portable Common Tool Environment Project	Russel Sandberg Design and Implementation of NFS	John Richards Software for a Graphics Terminal in C
10:00	Winfried Dulz System Management for a Distributed UNIX Environment	William Fraser-Campbell Implementing the NFS on System V.2	M. Guarducci Fiore Project: Wide Band Metropolitan Area Network
10:30	COFFEE	COFFEE	COFFEE
11:00	AFUU (France) News	GUUG (Germany) News	UKUUG (UK) News
11:10	Tom Killian Computer Music under Unix Eighth Edition	Pete Delaney A Guided Tour of OSI based NFS	Robert Heath Adding Commercial Data Communications to UNIX
11:40	Brian Collins The Design Of A Syntax-directed Text Editor	Peter Weinberger The Eighth Edition Remote Filesystem	Dave Presotto Matchmaker: The Eighth Edition Connection Server
12:10	LUNCH	LUNCH	LUNCH
14:00	DKUUG (Denmark) News	IUUG (Ireland) News	UNIGS (Switzerland) News
14:10	Philip Peake Implementing UNIX standards	Andy Rifkin RFS in System V.3	Bill Joy UNIX Workstations: The Next Four Years
14:40	Charles Bigelow Florentine Inventors Of Modern Alphabets C. Brisbois SIGMINI Information Management System	Lauren Weinstein Project Stargate	Mike Hawley Developments at Lucasfilm
15:30	AFTERNOON TEA	AFTERNOON TEA	AFTERNOON TEA
16:00	EUUG-S (Sweden) News	NLUUG (Netherlands) News	UUGA (Austria) News
16:10	Malcolm Agnew DB++ Database Management System	Roberto Novarese An Office Automation Solution with UNIX/MS-DOS	S. Mecenate The IBM 6150 Executive AIX
16:40	David Tilbrook Managing a Large Software Distribution	Marco Mercinelli SNAP: Restarting a 4.2 BSD process from a snapshot	Antonio Buongiorno Office Data Base Services in a UNIX Architecture
17:10		EUUG Business Meeting	

TECHNICAL PROGRAMME CONTENTS

ABSTRACTS FOR THE TECHNICAL PROGRAMME

Malcolm Agnew

DB++ Database Management System

Mail: robert@hslrswi
Phone: +69 597 029798

Concept ASA GmbH
Wolfsgangstrasse 6
D-6000 Frankfurt am Mainl
WEST GERMANY

The DB++ family of programs together comprise an efficient, flexible and reliable relational database management system for use with UNIX.

This paper discusses how the DB++ programs have been fully integrated into the UNIX framework. It then goes on to explain the choice of query language in addition to some of the unusual implementation details.

Charles Bigelow

Florentine Inventors Of Modern Alphabets

Mail: ukc!cab@su-ai.arpa
Phone: +1 415 788 8973

Bigelow and Holmes
15 Vandewater Street
San Francisco
CA 94133
USA

Bitmap screen displays and laser printers have enriched the appearance of computer literacy. Procrustean limitations of mono-case and mono-space that formerly degraded computer-produced text have been abolished. We now can enjoy the luxury of reading and printing text in lower-case as well as in capitals, in italic and bold styles as well as in roman, in proportionally-spaced fonts of different sizes as well as in monospaced fonts of a single size, in justified as well as in ragged-right columns.

Massimo Bologna

The Portable Common Tool Environment Project

Phone: +39 50 500211
Mail: 3bicoa!flor@iconet.uucp

The Portable Common Tool Environment (PCTE) project is carried out as part of the ESPRIT programme of the Commission of European Community.

The project aims at the definition and the implementation of a common framework, within which various software tools can be developed and integrated in order to provide a complete environment for software engineering.

The two main goals of the project are the portability across a wide range of machines and the compatibility for assuring a smooth transition from existing software development practices.

The technical approach to portability is discussed in this paper: UNIX is the means by which PCTE will be widely available; the PCTE basic mechanisms are built on top of UNIX: in fact PCTE can be seen as an extension to UNIX in the area of distribution, friendly user interfaces and database for software engineering.

This last aspect is described in this paper: functional as well as implementation aspects will be dealt with.

Finally, tools developed for the PCTE database are described and discussed.

C. Brisbois

SIGMINI Information Management System

Union Miniere SA
Division Information
Avenue Louise 54, BTE 10
B-1050 Brussels
BELGIUM

Sigmini is designed to process heterogeneous information, including quantities, which have complex interrelationships. It is related to both database systems and classical documentation retrieval systems. In either case, Sigmini is designed to avoid the necessity to declare in advance the data or relationships.

Antonio Buongiorno

Office Data Base Services in a UNIX Architecture

Phone: +39 125 52 15 92

Olivetti DSRI/DPS
IVREA
Italy

Antonio Buongiorno
Franco Calvo
Bruno Pepino

Classification, filing and retrieval are important but time-expensive activities in an office environment. Attempts to reduce times, and thus costs, involved in these processes and to increase the effectiveness of an retrieval system are rapidly gaining importance for a better information mamagement in an office. The paper outlines a solution for filing and retrieving "objects" in an architecture based on UNIX servers and networks of personal computers. The focus is mainly on the data model that supports the Office Data Base and allows a very fast retrieval of structured and non structured information.

In the first section is described a general architecture, in the second the functionalities and the data model and in the third the prototype package AMEDEUS.

Brian Collins

The Design Of A Syntax-directed Text Editor

Mail: ukc!bco@ist
Phone: +44 1 581 8155

Imperial Software Technology
60 Albert Court
Prince Consort Road
London
SW7 2BH

The editing and user-interface system described in this paper presents many different guises to the user. Among these are:

1) A general-purpose, multi-file, multi-windowing screen editor for ASCII text files and simple terminals.
2) A forms system for constrained data entry and presentation.
3) An interactive windowing front-end to application programs.
4) A syntax-directed editor.

This paper concentrates on the latter.

In the design of such an editor, the choice of facilities offered to the user is often more difficult than the actual implementation. In this editor, the decision taken was to make the interface appear to the user as close as possible to a standard text editor, hence the more descriptive title of 'syntax-directed text editor'.

Normal text editing operations can be seen to fall into two classes: those which only affect one line of text, and those which affect a number of lines. The user is allowed to edit freely any text within a line, using text-editing operations. On leaving the line, the edited text is re-parsed and errors corrected or reported. Editing operations which insert, delete, move or copy lines must explicitly maintain syntactic correctness across line boundaries. For example, a line containing 'END' could be automatically added when a 'BEGIN' is detected.

The overall effect is to provide a text editor in which it is impossible to enter a syntactically incorrect program.

The editor is language-independent (indeed, it may handle many different languages simultaneously). A language may be supported if it conforms to a few obvious restrictions (similar to those of lex & yacc). The syntax of the language is described in an extended, annotated BNF, which also specifies the layout rules for line breaks, spaces and indentation.

The paper provides a description of the editor, both from the user's view and from the language implementor's. It describes how the syntactic structure is maintained in what is essentially a text editor, and the techniques used for the fast re-parsing of edited text. Finally, the paper assesses the applicability of syntax-directed editing to various languages, its use in a programming environment, and a specific analysis of the problems in the syntax of the language C.

Pete Delaney
A Guided Tour of OSI based NFS

Mail: ukc!ecrcvax!pete@unido.uucp
Phone: +49 89 92699 138

Rockey Mountain UNIX Cons
ECRC
Arabellastrasse 17
D-8000
Munchen 81
WEST GERMANY

A guided tour of the ESPRIT OSI implementation on 4.2BSD will be presented. An example of mounting a filesystem via Sun's NFS using OSI protocols over X25 will be demonstrated with:

- A Kernel trace showing major events
- A diagram of cronological events, with references to the kernel trace.
- Topological organization of network structures.

Due to the lengthy nature of the trace, copies will be distributed to the audience. Implementation details and divergencies from the US NBS and General Motors MAP project will be discussed.

The merits and pitfalls of the OSI protocols will be presented along with recomendations for further work.

Winfried Dulz
System Management for a Distributed UNIX Environment

Mail: ukc!fauern!faui70!dulz@unido.uucp
Phone: +49 9131 857929

IMMD 7
Martensstrasse 3
8520 Erlangen
WEST GERMANY

In analogy to system management for central server configurations, distributed systems must also provide utilities for user-handling, system- accounting and resource-accessing. We show for the UNIX network operating system Newcastle Connection how transaction-oriented protocols based on communicating client/server processes can solve this class of problems. To that end all local user-files /etc/passwd are concatenated to a system-wide database that also contains information about login hosts and UNIX systems that can be reached by means of the Newcastle Connection. The only process that may update this database is a reliable and redundant system process that guarantees the necessary data consistency.

William Fraser-Campbell

Implementing the NFS on System V.2

Mail: ukc!inset!bill
Phone: +44 1 482 2525

The Instruction Set Ltd
152-156 Kentish Town Road
London
NW1 9QB

System V has been enhanced to support multiple file system types using a common interface within the kernel.

Using the Sun Network File System architecture, our implementation supports System V files on local disks, and acts as both client and server for network file systems using published NFS protocols.

This paper will present details of the implementation.

M Guarducci

Fiore Project: Wide Band Metropolitan Area Network

Electronics Department
Florence University
Florence
ITALY

The design and implementation study discussed in this paper outlines first problems and solutions of the design work, followed by implementation strategies of services supplied to final users: file transfer, messages, electronic mail.

The realisation foresees use of host computers running the UNIX operating system connected on the wide band based MAN of the Fiore project in Florence based on th Localnet 20 protocol by Sytek, on which Ethernet LAN's and stand-alone computers may be connected.

Mike Hawley

Developments at Lucasfilm

Mail: ukc!mike@dagobah
Phone: +1 415 485 5000

PO Box CS 8180
San Rafael
CA 94912
USA

Mike will discuss UNIX & computers at Lucasfilm. The excitement comes from combining information technology with the richest possible kinds of communication media. Examples range from the high-end graphics and audio work done there (with the PIXAR and ASP systems) to earthier projects involving large databases of sound effects, books, poetry, etc, and of course, music.

Most of the work exemplifies UNIX applications and systems development with an artistic bent.

Robert Heath

Adding Commercial Data Communications to UNIX

Phone: +1 513 445 6583

NCR Corp.
Columbia, South Carolina

Tlx: 851295467CZ8123Z 295467

As the UNIX operating system becomes more widespread in small business systems, the need to add commercial data communications becomes important. This paper discusses how NCR in its UNIX-based supermicrocomputer, the Tower, has supplemented the basic data communications tools provided by AT&T with both industry-standard and internationally standard protocols. The resulting product provides interconnectability in three general areas: asynchronous, synchronous, and local area networking. The paper illustrates which protocols are important for a general-purpose small business system.

The well-known Call UNIX (CU) and UUCP are standard Tower utilities for asynchronous networking. A high-performance, asynchronous adapter, which moves the tty driver off the main processor for both networking and workstation control is described.

UNIX System V is deficient in synchronous protocols. Described is an intelligent, synchronous adapter which supports multiple, block-oriented protocols such as SDLC, HDLC, and Bisync. Data link control drivers were added to the kernel to complement standard networking packages such as SNA, X.25, and 2780/3780 Bisync, and 3270 Bisync. The paper describes how 3270 screen emulations reuse UNIX concepts such as termcap and remote job entry emulations reuse the printer spooler.

Tower local area networking (Towernet) is provided through a programmable Ethernet adapter which offloads time-critical operations. Towernet services described include electronic mail, file transfer, virtual terminal, PC interconnection, and wide-area networking. Another lan-based option is the distributed resource system which not only implements a distributed file system but also provides transparent access of remote devices.

This paper illustrates how modern, layered protocols are distributed within UNIX. It details particular problems in intertask communications and multiplexing separate data streams. Solutions to these problems are presented in application software, kernel software, firmware, and hardware. Strategies for network management, diagnostics, and maintenance are offered.

Bill Joy

UNIX Workstations: The Next Four Years

Mail: ukc!inset!sunuk!sun!wnj
Phone: +1 415 960 1300

Sun Microsystems Inc.
2550 Garcia Avenue
Mountain View
CA 94043
USA

At the EUUG conference in Paris 1982 Bill predicted the future of UNIX workstations and the technology associated with them for the three years which were to follow. That time has elapsed and therefore he will update that talk with further insights into the developments in the next few years.

The talk will not only cover UNIX workstations, but developments in the technology applicable, regardless of operating system.

Tom Killian

Computer Music under UNIX Eighth Edition

Mail: ukc!tom@ikeya
Phone: +1 201 582 3000

AT&T Bell Laboratories
Murray Hill
New Jersey
USA

We describe an evolving computer music system which draws upon many of the novel facilities of the 8th edition as well as the standard repertoire of Unix tools. The Teletype 5620 bitmap display serves both as the user's terminal and real-time controller. The mux window system is used to download a MIDI interface driver which services other windows (by direct code sharing) and host processes (which write on the driver's control stream). We presently have two MIDI-compatible instruments, a Yamaha DX7 and TX816.

Window programs include a piano-roll style score facility and a virtual keyboard. Host programs include a music compiler, "m," which converts an ASCII score notation into MIDI events; it is based on lex and yacc, making it very easy to develop in response to user needs. There is also a variety of filters which perform simple transformations (e.g., time and pitch translation) on MIDI files. The latter are ASCII, so that, for example, output from the DX7 keyboard can be translated into "m" notation with an awk script, and other Unix text filters (especially sed and sort) and "c" programs are useful as well.

We will show (and play) examples of pieces written in "m," "c," and the Bourne shell, and discuss future plans which center around 12-tone serial composition.

Tom Killian began his career as an experimental high- energy physicist at CERN and Brookhaven National Laboratory. Frustrated in his attempts to persuade his colleagues of the evils of JCL, he eventually found his way to Bell Labs, where he has been a member of the Computing Science Research Center since 1982.

S Mecenate

The IBM 6150 Executive AIX

Phone: +44 1 995 1441

IBM UK Ltd
Chiswick
London W4

In January 1986 IBM has announced the IBM 6150 micro computer with the Advanced Interactive Executive (AIX) operating system.

As the base for AIX IBM chose AT&T's UNIX System V because it provides considerable functional power to the individual user, multi-user capabilities, is open-ended, and has a large user and application base.

However in choosing UNIX IBM recognised the need to make significant extensions and enhancements to meet the needs of our expected customers and their applications.

The presentation will discuss some of these major modifications and additions.

Marco Mercinelli

SNAP: Restarting a 4.2 BSD process from a snapshot

Mail: ukc!cselt!marco@i2unix.uucp

Sezione Metodologie Software
Divisione Informatica
Centro Studi e Laboratori Telecomunicazioni
10148 Torino
ITALY

SNAP is an extension to UNIX4.2BSD for taking a snapshot of a running process and restarting its execution at a later time.

A snapshot is composed of a process core image and information about its execution environment (opened files, devices, tty settings, etc.). It is not necessary to kill the process for taking the snapshot.

A process can be restarted at any time, even after a system crash. Its execution continue at the point the snapshot was taken in a "quasi" transparent fashion. All the process resources should be available and are set to a suitable state.

The snapshot facility can be used as a building block for several higher level mechanisms such as crash recovery, debugging, backtracking and process migration.

The current implementation of SNAP can only restart a single process with no IPC connections. Further work is needed in order to extend SNAP for managing groups of related processes and connections in a distributed environment.

Roberto Novarese

An Office Automation Solution with UNIX/MS-DOS

Phone: +39 125 52 15 92

Olivetti DSRI/DPS
Ivrea
ITALY

In this paper a set of Office Automation requirements of the Economic European Community are presented. A solution that has been designed and prototyped by Olivetti is then discussed. The proposal is based on the integration of UNIX mini and MS-DOS personal computers on a local area network. A set of integrated Office Productivity Tools on the workstations provide the support to professional and secretarial activities.

Personal Computers Support Services provide the sharing of resources (file server, print server and communication server functionalities). An X.400 Electronic Mail and an Archiving System are the Office Cooperation Services designed for the integration of this soluction in a Multivendor Architecture.

Philip Peake

Implementing UNIX standards

Mail: ukc!philip@axis
Phone: +33 1 4603 3775

Axis Digital
135 Rue D'Aguesseau
92100 Boúlogne
FRANCE

As UNIX becomes more firmly established in the commercial computing world there is much pressure, and resulting action for standardisation. For example, the SVID and X/OPEN publications.

This presentation looks at some of the problems encountered during the development, and subsequent porting of applications which either run on, or communicate with UNIX systems. Some attention is also paid to the existing standards; as found in practice, and as proposed in the above documents.

Dave Presotto

Matchmaker: The Eighth Edition Connection Server

Mail: ukc!presotto@research
Phone: +1 201 582 5213

AT&T Bell Laboratories
Murray Hill
New Jersey 07974
USA

Matchmaker is a connection service for Eighth Edition UNIX. Using Matchmaker, processes can connect to processes on the same system or across a variety of networks. Unlike other solutions to this problem, such as 4.2 BSD's sockets, ours separates network protocols and communication properties to such an extent that application programs using it need not be cognizant of the network or network protocol on which the connection is built.

Matchmaker is based on Dennis Ritchie's Streams, a mechanism for providing two way byte streams between processes and devices or between processes and other processes. The unique properties of Streams make Matchmaker possible. Using Streams we can perform functions such as circuit setup, circuit shutdown, and data stream processing in the kernel, in processes, or even in separate processors as the situation dictates.

John Richards

Software for a Graphics Terminal in C

Mail: ucl-cs!richards@uk.ac.bristol.qvc
Phone: +44 272 303030

University of Bristol Computer Centre
University Walk
Bristol
BS8 1TW

This paper describes the development of software for incorporation in a new intelligent raster graphics terminal. The terminal provides support for windows and graphics segments. The software was written in C and developed and tested on a UNIX^{TM} system before being placed in ROM in the terminal. The paper shows how considerable use was made of structures and the storage allocation functions to provide a generalised segment storage scheme. Examples are given of the way language constructs were used to obtain fast, but portable, code. The finished software was ported to the terminal with hardly any modifications.

Andy Rifkin

RFS in System V.3

Mail: ukc!uel!attunix!sfjec!apr
Phone: +1 201 522 6283

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

Andy is a principal developer of AT&T's distributed UNIX file system known as RFS. He joined AT&T after receiving his masters degree in computer science from Cornell University.

The AT&T distributed file system known as RFS, provides users with transparent access to remote filesystems. The goal behind RFS is to offer the complete functionality of the UNIX filesystem (i.e., special devices, record locking, named pipes) without compromising the UNIX filesystem semantics. That is, programs which run in a single machine environment will run in an RFS environment with no change.

The implementation of RFS is done at the kernel level, using the standard UNIX mount command to access remote resources. RFS was designed to be both protocol and media independent using the Streams architecture available in UNIX System V Release 3.

This talk will describe the RFS architecture from both a communication and kernel perspective. In addition a comparison between RFS and other available distributed file systems will be made in order to highlight their differences.

Russel Sandberg

Design and Implementation of NFS

Mail: ukc!inset!sunuk!sun!phoenix!rusty
Phone: +1 415 960 1300

Sun Microsystems Inc
2550 Garcia Avenue
Mountain View
CA 94043
USA

The Sun NFS provides transparent, remote access to filesystems. Unlike other remote filesystems available for UNIX, NFS is designed to be easily portable to other operating systems and architictures.

This paper describes the design and implementation of NFS along with some experience of porting it to other systems.

David Tilbrook

Managing a Large Software Distribution

Mail: ukc!dt@ist
Phone: +44 1 581 8155

Imperial Software Technology
60 Albert Court
Prince Consort Road
London
SW7 2BH

One of the major problems at IST is the management of more than eight mega-byte of software and related data for IST's own machines and those of our clients. For obvious reasons it is desirable to centralize the management of the source in a single location and to extract the distributions as required. This presents a variety of problems, principally with respect to the variations in the target systems. This paper discusses the solutions developed to overcome the differences between operating systems, interdependcies within the source, the variations in the available software tools.

Peter Weinberger

The Eighth Edition Remote Filesystem

Mail: ukc!pjw@seki
Phone: +1 201 582 3000 ex: 7214

AT&T Bell Laboratories
Murray Hill
NJ
USA

Peter is famous for at least two pieces of work. Firstly, he is the W in AWK (Aho, Weinberger and Kernighan), that useful pattern matching and scanning language the we all use daily. His second well known work is in the area of distributed filesystems, in particular he was responsible for the initial design and implementation of the Edition VIII remote filesystem.

In Florence, Peter will explain the motivation behind this work on the remote filesystem and study the model of the world it assumes.

Lauren Weinstein

Project Stargate

Mail: ukc!lauren@vortex
Phone: +1 213 645 7200

Computer/Telecommunications Consultant
PO BOX 2284
Culver City
CA 90231
USA

This paper and talk will review the current status of the ongoing "Stargate" project, which is experimenting with the transmission of "netnews"-type materials over the satellite vertical broadcasting interval of television "Superstation" WTBS, a very widely available basic cable television service in the United States. The techniques used allow the Stargate data to exist in parallel with the standard video and audio of the television operation. Satellite-delivered WTBS is currently available to over 33 million cable subscribers (households and businesses) throughout the United States, and is also received directly by privately owned satellite earthstations. This paper discusses both the technical and non-technical (i.e. organizational, content, policy, etc.) aspects of the project.

The DB++ Relational Database Management System

Malcolm Agnew
Concept ASA GmbH,

Wolfsgangstrasse 6,
D-6000 Frankfurt am Main 1, West Germany

and

J. Robert Ward
Hasler AG,

Belpstrasse 23, CH-3000 Berne 14, Switzerland

UUCP : ... {seismo,decvax,ukc, ... }!mcvax!cernvax!hslrswi!robert
EDU: hslrswi!robert%cernvax.bitnet@UCBJADE.Berkeley.EDU
ARPA: hslrswi!robert%cernvax.bitnet@WISCVM.ARPA

ABSTRACT

The db++ relational database management system comprises a series of programs that provide a general yet efficient data processing tool for the Unix [1] environment. It is currently implemented on a wide range of machines and Unix operating systems.

This paper discusses how the db++ system has achieved flexibility through being integrated it into the Unix working environment as well as possible. It then describes the system's design and efficient implementation. In particular, it describes the method of data storage on disc and how a powerful pipe-line algorithm aids the query processor.

1. Introduction

The db++ family of programs comprise an efficient, flexible and reliable relational database management system. This series of programs is of interest for two primary reasons. First, the db++ system supports features that are generally not to be found on other commercially available database programs running under Unix - these design features allow the db++ programs to be easily combined with other Unix utilities and also permit one to formulate complicated queries to the DBMS. Second, the db++ query processor and editor programs use unique internal algorithms to ensure that complicated queries can be processed efficiently and safely.

This paper discusses how the db++ programs have been fully integrated into the Unix framework. It then goes on to explain the choice of some of the unusual implementation details.

[1] Unix is a trademark of AT&T Bell Laboratories.

2. Integration Within Unix

The db++ system was designed to be flexible in that it had to be able to meet a wide range of applications. This goal has been achieved by integrating the system into the framework of Unix as much as possible. The choice of a powerful query language has also proved to be important.

Unlike most commercially available database systems such as Oracle or Ingres, the db++ system does not impose a new working environment on the user. Db++ was developed to be a standard computing tool and can be combined with other Unix utilities. For instance, the output from the query processor can be piped to standard programs such as awk(1) or sed(1). This aspect immediately increases the system's flexibility and usefulness.

There is no "hidden place" where data is stored. Nor is there any concept of a "system catalogue". All information concerning a single relation is contained within one binary file. This implies it is possible to use standard Unix commands such as cp(1), chmod(1) or tar(1) on individual relation files without having to learn new commands. For instance, one can delete a relation simply by using the standard rm(1) command. There is no need to provide a special command that would have to delete the stored data and then ensure that the relation's entry in a global system catalogue is also removed. This approach simplifies the implementation and the effort involved in learning the system.

Thus each one of the db++ programs follows the "Unix philosophy" of cooperating well with the existing environment.

2.1. Library of Raccess routines.

The db++ programs are built around a single set of access method routines, the Raccess library. These routines allow one to open a given relation file, to create a new relation file and to insert and extract data from such files. They also support higher-level functions on stored data. For instance, one may call these routines to sort stored data, to create a new secondary index for a relation file, to locate an item of stored data quickly or to undo an extensive transaction on a relation file [2].

2.2. Piping of Relational Data.

The db++ programs allow relational data to be sent through Unix pipes. This fact also enhances their integration within the Unix environment. One may, for instance, write a specialised program using the Raccess library routines to read and manipulate relational data in some way. This program can then be combined with the db++ query processor program in order to perform additional processing on the data.

2.3. Current Applications.

A description of a few of the current applications to which the db++ programs have been applied may help to give some idea of the system's flexibility.

One application illustrates how the db++ system can be combined with a command pre-processor. This application is the development of a database storing employee records. Because of the lack of a hidden environment, it was a straightforward matter to implement a csh(1) command pre-processor on top of the db++ programs. The command pre-processor hides many details of the database implementation from those who maintain the data.

[2] The Raccess library routines are also implemented on the MS-DOS operating system.

A second example illustrates how the db++ programs can perform complex queries on stored data. This application was the creation of a financial book-keeping system for an organisation with some 300 employees. The db++ programs are invoked each month to produce reports summarising the movements on a set of accounts, to perform tax calculations and so on. Processing the input data involves queries with over 80 relational operators.

A third application is a retail stock management system for pharmacists. The system is designed with three primary objectives : first, to optimise warehouse storage by taking account of turnover frequency as well as expiring dates and automating ordering ; second, to provide relevant online information on pharmaceutical products ; third, to provide general business facilities such as book-keeping and text processing. Information on some 200 thousand pharmaceutical products is held in two relations that are accessed frequently each day. The remainder of the database is held in a further twenty or so relations of cardinality 10 to 10000. This application contrasts sharply with the previous two in that most accesses to the database are made from mask driven application programs built with the Raccess library routines.

It is also worth noting that other, groups are making use of the openness of db++ to make their own extensions to the system. For instance, the Universities of York and Newcastle are implementing Codd's extended relational model RM/T [CODD79] on top of db++ for use in an Alvey financed project. In other words, this application uses the db++ system as a "database engine".

The University of Manchester is similarly involved in an Alvey project on VSLI design in which they intend to build on db++ to create a CAD database that supports hierarchical views of objects.

2.4. The Db++ Programs.

The db++ family is broken up into logically distinct programs. The various programs are briefly described as follows -

Dbcreate reads ASCII data from the standard input and converts it to the special file format required by the other db++ programs. This is generally the first step in setting up a database.

Dbappend is used to place additional ASCII data into a relation file.

Dbls reports information about the data stored in a database. For instance, it reports the degree of a relation as well as the names and data-types of each domain. It is the db++ equivalent of the standard ls(1) command.

Db is the main query processor command. Db may be instructed to list data from a database or to combine and transform existing data to produce new output. This program is described in greater detail below.

Dbedit allows one to change the data stored in a relation file. It is an interactive program and has a command syntax reminiscent of ex(1). Dbedit allows one to make extensive changes to a relation using its :global command, or changes that are based on existing data values.

Dbmodify reformats a relation file. This may be necessary if new key domains are to be defined for a relation file. It may also be used to specify one or more secondary indices for a relation.

Dbrpg is a report generator program. It is the db++ equivalent of the standard awk(1) program : it takes a relation file and prints its contents according to a given command script.

A further two systems are available which, although not forming an integral part of the DBMS, are helpful in building applications.

Dbmask is a set of programs and routines that simplify data entry as well as formulating queries and displaying results according to a mask.

Dbmenu is a menu shell that can replace the usual C shell or Bourne shell for particular applications.

2.5. Command Language.

A retrieval command to the db query processor program is phrased in a language based on Codd's relational algebra [CODD72]. The generality of the query language allows both simple and complicated queries. This is important since it allows the user to formulate casual one-off queries easily. On the other hand, it is possible to develop a query instructing db to evaluate a complicated relational expression.

Designers of relational DBMS's seem to have concentrated on the relational calculus rather than the algebra. (Relational programming languages based on the calculus include SQL [ASTR76] and QUEL [STON75]). The predominance of such systems may well be a historical accident, rather than a deliberate technical decision. The fact that SQL is the standard query language for IBM's database products has also promoted its acceptance.

However, we agree with King [KING79] and [YORK85] who state that the algebra is more adept in developing complicated queries. The db++ system allows for complicated query processing on stored data. Therefore, the algebra was chosen as the basis for the query language.

Unlike Codd's original relational algebraic language, the db query language is terse. Db uses punctuation characters to signify relational operators. Indeed, the query language is modelled after the C programming language, especially with regard to the way in which scalar operators and scalar typing rules are defined. (This query language is described fully in [CONC85]). For example, the following query -

```
Cars :: make == "Volkswagen"
```

means 'list tuples from relation file Cars such that fields from domain 'make' have the value "Volkswagen"'. Similarly, this query -

```
Cars ** Manufacturers
```

means 'list the join of relations Cars and Manufacturers.' (Here, the join is over domains having the same names in both source relations). Besides Codd's basic relational operators, the db program includes an aggregation operator. This is a generalised operator by which one can find maxima, minima, totals, and so on, from a relation. It allows one to split up a relation into groups of tuples, each group having the same value in one or more domains, and to perform some summarising operation on each group. (This is similar to the operation of 'glumping' described in [HITC77]). Db also provides a sort operator. This is useful to order a large listing of a relation. For instance, this query -

```
Cars ## make price
```

means "sort relation Cars over domains 'make' and 'price'".

The query language allows one to evaluate relational expressions and assign them to new output files. The output file may be created in relational binary format or as an ASCII file. For instance, this query evaluates the same relational expression as the one above, but places the output in a new relation file called 'Result' -

```
Result <= Cars ## make price
```

Db also allows one to call upon the standard C language procedure printf to format listed output. For instance, the following query lists the output fields according to the given string -

```
Result "make = %s, price = %ld\n" <== Cars ## make price
```

3. Implementation of the Access Methods

The Access methods rely on each relation file having a certain binary structure. This is somewhat analogous to the structure of a Unix file-system. Although most users are unaware and unconcerned with the underlying implementation details, such details have a significant bearing on the higher-level facilities offered by these programs.

A single relation file contains the stored data itself as well as information describing that data. Secondary index information is also stored within the same file.

3.1. The Super-Page.

The first block of a relation file holds information describing the data. For instance, the cardinality of the relation, its degree and other information are held in this block. Information pertaining to each domain, for instance its name and data-type, is also held in this first block. In the case of sorted relations, ordering information is also held here. This first block, therefore, is crucial to accessing the data contained in the rest of the file. It is analogous to the super-block of a Unix file-system. In db++ terminology, it is termed a "super-page".

3.2. Storage Modes.

The db++ programs support two types of relation file. The data contained in a Heap relation is unordered. It is not possible to optimise access to such relations and so a search for a specific item of data involves scanning through the entire relation file. On the other hand, there is a low update cost associated with adding or removing data from such a relation. Furthermore, the db++ programs recognise when relational data is to be sent to a non-disc device, such as a Unix pipe or other special device. When this happens, the data is sent as a Heap relation.

The other type of relation is termed a B+-tree relation. The B+-tree data structure has become the "de facto" standard among database systems because of its low storage overheads and guaranteed low update costs, although updating a B+-tree relation is usually somewhat more expensive than updating a heap relation.

The data held in a B+-tree relation is always ordered in some specific way and this allows the db++ programs to optimise access to specific items of data. These programs can also take other advantages of a B+-tree relation's ordering. For instance, this ordering information can be used to optimise processing of the sort, join, intersection and difference operators of the query processor. Finally, because of the random access nature of a B+-tree relation, it cannot be passed through a Unix pipe or other non-seekable device. Because of the general usefulness of the B+-tree storage mode, the db++ programs create new relation files with the B+-tree storage mode unless instructed otherwise.

Regardless of the chosen storage mode, the relation file is divided into "pages", with each page occupying a small multiple of the underlying file-system's block size. For instance, the 4.2bsd Vax implementation of the db++ programs uses a page size of 4096 bytes. Except for the super-page, a data page is the unit of I/O.

3.3. Secondary Indices

Data within a B+-tree relation file is held in a series of B+-tree data structures with one such B+-tree for each primary or secondary index of the relation. In other words, there is a complete copy of the stored data for each index of the relation. Additional disc storage is therefore traded for increased efficiency.

Each such B+-tree structure can be uniquely identified by specifying the location of its root page, the locations of its first and last leaf pages, by the number of associated key domains and by the associated ordering information. This information is also held in the super-page.

3.4. Transaction Processing.

A database management system must support some transaction facility so that stored data remains in a consistent state even if a process crashes while updating the stored data. For instance, a process updating data may crash if the machine itself crashes, if the process is interrupted or if some other hard error occurs.

The db++ programs support such a transaction facility for B+-tree relations only. This facility is also available to programmers using the Raccess routines, thus permitting "rollback" points to be set in an application program.

This transaction facility is perceived by the user as follows. Whenever a query is run, one of two outcomes may happen. Either the query runs to completion successfully, in which case all updated data is written back to the relation file and the relation is closed normally, or the query is aborted in which case it appears that no changes have been made to the relation. This transaction facility is independent of the size of the relation file as well as the nature of the changes. Furthermore, when a transaction is aborted, there is an immediate and automatic roll-back to the previous state of the relation file.

This transaction facility is implemented as follows. Whenever an item of data is to be updated, the first action is to locate the corresponding leaf page from the B+-tree. If this is the first modification to the leaf page since the relation was opened, the page is copied to a free page in the disc file and then the copy is updated. However, because the leaf page has effectively changed its location within the relation file, the index page of the B+-tree that references the leaf page (the parent of the leaf page) must also be changed so that it references the new leaf copy.

The same logic then applies to the index page. If this is the first time that the index page is being changed since the relation was opened, the index page is copied to a new location on the disc. If the B+-tree has a depth greater than 2 - that is, if the index page itself has a parent - then that parent page too is reallocated and updated.

This reallocation of pages propagates itself up to the root page at the top of the B+-tree structure. It is important to note that the original pages are never changed in any way. Therefore, if the process updating the relation file crashes in any way, the original data is still available.

This operation is illustrated in figure 1. The resulting B+-tree structure is depicted in figure 2.

The problem, therefore, is to devise a system for keeping track of which pages have been reallocated - in other words, which pages hold the updated information - and which belong to the original state of the relation file. This is accomplished using a series of bit-maps that are scattered though the relation file. Each such bit-map occupies an entire data page.

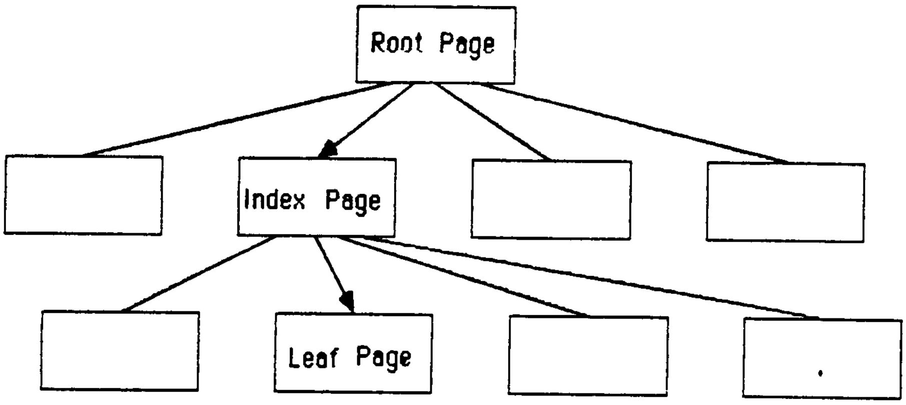


Fig. 1. This illustration depicts a B+-tree of order 3 containing a leaf page, an index page and a root page. When a tuple is to be added or deleted from the given leaf page, and the leaf page has not been updated since the relation was opened, the leaf page is first copied on to a fresh, unallocated page so that the original data is not disturbed. However, because the leaf page has changed its location in the relation, its parent page, the index page, must also be updated. Again, if this is the first time that the index page is being changed, it too must be copied. This process repeats itself until the root page of the B+-tree is also reallocated. The resulting B+-tree structure is depicted in figure 2.

Each bit in a map describes whether a corresponding page in the relation is currently allocated to holding data or whether it is free - that is, whether it can be reused. In fact, there are two parallel bit-maps. The first bit-map holds the state of the relation file when it is first opened. This bit-map is never changed during a transaction. The second bit-map describes the current, new state of the relation when it is being updated. This second bit-map is updated while the data pages themselves are reallocated.

Given this information, it is a straightforward matter to locate a free, unused page in the relation file that can be used for storing data or that can be used to hold a reallocated data or index page. When the relation file is opened, the bit-map showing the current page allocation is copied to its parallel bit-map describing the updated allocation. Whenever a new page is to be allocated, the two bit-maps are logically ORed together. A page is deemed to be reusable if it is marked as being free on both bit-maps. That is, a page may only be reused if it is marked as being available on both the original and current page allocations. This operation is illustrated in figure 3. If there are no reusable pages, the relation is extended by allocating a new page at the end of the relation file.

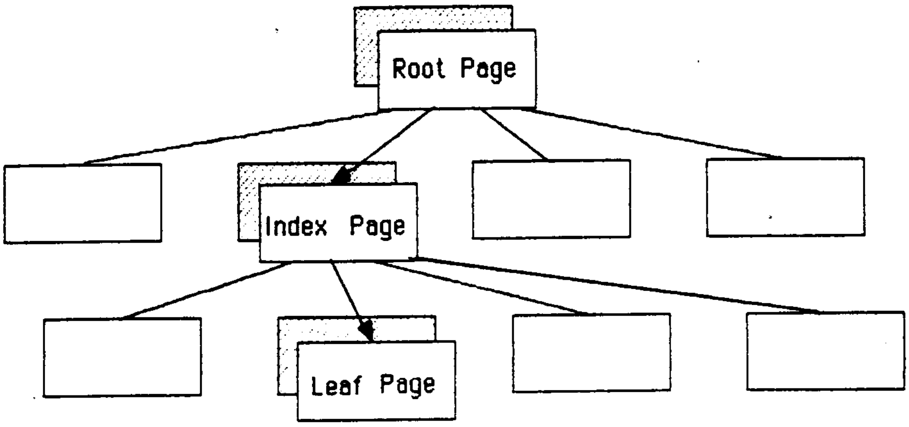
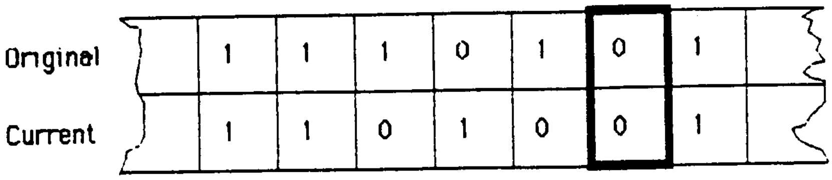


Fig. 2. The illustration below depicts the B+-tree of figure 1 after the pages leading up to the root page have all been reallocated. The original data is still present and can be retrieved if the updating process crashes.



Original		1	1	1	0	1	0	1
Current		1	1	0	1	0	0	1

Fig. 3. Here we consider a search through the bit-map when a page is to be reallocated. The bit-map corresponding to the original page allocation is ORed with that corresponding to the current allocation. In this example, the page corresponding to the highlighted bit positions is chosen as being free because all the pages up until that one are claimed by either the current or original allocations.

If the transaction is aborted, there is no logical effect on the relation file because all the original information, including the original bit-map state, is still present.

If the transaction runs to completion, the super-page, which also contains a single bit describing which is to be considered the most recent state of the relation, is written back to disc. The one remaining problem, therefore, is to consider what happens if this write to the super-page fails. The size of the super-page is chosen to be the same as a single block of the underlying hardware. On most Unix systems this is 1024 bytes. Therefore, only a single write operation is required to mark the entire relation file as being updated. If the system crashes even in the middle of that write, it is to be hoped that most disc controllers would continue with the operation. The transaction facility, therefore, relies on the updating of the super-page being an atomic operation.

This algorithm implies that the db++ programs do not attempt to localise information on disc across successive page allocations. All requests for I/O are performed through the normal read(2), write(2) and lseek(2) system calls using offsets measured from the beginning of the file. Because Unix itself, however, guarantees no locality of successive pages written out to disc, it is impossible for the db++ programs to attempt to optimise disc access from one page to another. The fact that Unix is a multi-tasking system also complicates the problem of locality. Therefore, we have deliberately chosen to ignore this issue [3].

This algorithm, then, is well suited to a database management system. Not only are,its effects well defined, but it also has a cheap overhead in terms of disc storage and processor time. Finally, an extension of this algorithm has facilitated the implementation of the :undo command of the dbedit program.

3.5. Implementation of the Undo Facility

The dbedit program supports an :undo command, the effects of which are to restore a relation file to the state it was in before a previous editing command took effect. The dbedit program does not copy a relation file before editing commences, thereby making it possible to edit large relation files with a low initial overhead. Rather, it appears that a copy is being edited ; no changes are made to the relation file until committed with an explicit :write command.

This is implemented as a generalisation of the transaction algorithm described above. Three bit-maps are used to describe the page allocations within the relation file. As before, the first and second bit-maps describe the initial and current page allocations. The third bit-map describes the page allocation of the relation file as it was just before the previous command was carried out. The implementation of the :undo command , therefore, just involves swapping the editor's idea of which is the current bit-map. Thus, the :undo command also has a low overhead.

Again, this facility is also available to programmers using the Raccess routines.

4. Implementation of the Query Processor

This section explains the implementation of the program db, the main query processor. A powerful pipe-line algorithm ensures that queries are processed efficiently. The algorithm avoids writing temporary results out to an intermediate disc file. The pipe-line is a recursive structure set up after parse time.

[3] Nevertheless, we cannot resist reporting that db++ performs considerably more efficiently in benchmark test than do some other commercial products claiming advantages from using raw disc I/O. This perhaps becomes clearer when one realises that many database operations are CPU and not disc bound. This is further discussed in [STON80].

4.1. Parsing The Query Input.

The parser was written with the aid of the YACC compiler-compiler [JOHN75]. The parser accepts the command-language and builds an in-core representation of the query. This is a tree structure containing only the names of domains and relations, the values of scalar constants and so on. The leaves of the parse tree correspond to source relations. Nodes within the tree correspond to relational operators.

4.2. Construction Of The Pipe-Line.

The next stage is to transform the basic tree formed by the parser into the pipeline structure. There is a one-to-one mapping from the nodes of the parse tree to those in the pipe-line. Every relational operator appearing in the query corresponds to a unique node in the pipe. Thus the pipe may be a single line or a tree. The nodes of the pipe contain more information than those in the parse tree. Each node holds, amongst other things, a description of the relation at that stage of the pipe. A recursive procedure is called upon to build the pipe-line. The leaves are the first to be transformed into pipe nodes and this involves opening the source relations. This transformation procedure is also responsible for initialising the pipe.

For example, it may be called upon to build a node corresponding to a relational selection. It then constructs an internal representation of the selection constraint. Figure 4 shows an example of a pipe-line corresponding to the selection and subsequent listing of tuples read from a source relation.

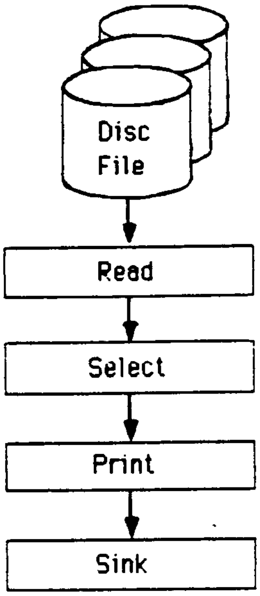


Fig. 4. The pipe-line depicted here corresponds to a selection and subsequent listing of tuples read from a source relation.

A further example may be seen in the case of the join operator. Here, the transformation routine determines the commonly named domains of the source relations. (The join operation is defined as being over those domains from the source relations that share the same names.) It initialises the node with information showing how the source tuples are to be compared and of how the output tuples are to be constructed. Finally, it forms a description of the resulting relation. One final operation is applied to the pipe. If the operator at the base of the pipe is not an explicit assignment to an output file, it is assumed that the relational expression is to be evaluated and listed to the standard output. An extra node corresponding to a print operation is then built into the root. Thus the construction of the pipe-line involves opening the source relations, identifying the format of the relation at each node and generally initialising internal data structures.

4.3. Processing Tuples.

Once the pipe-line is constructed, it is used to evaluate the specified relation expression. Apart from the information described above, each node of the pipe holds the address of a procedure to be applied to the data stored in that node. Each such procedure behaves as a coroutine. (The notion of a coroutine is discussed in [KNUT77]). Every implemented relational operator has a corresponding coroutine.

Most coroutines may be thought of as consumers and producers of tuples. A coroutine is generally activated by its parent coroutine. The parent requests one tuple from its child and then hangs, waiting for the request to be satisfied. A coroutine always has one parent but may have zero, one or two children, depending whether it corresponds to source relation or to a unary or binary relational operator.

The coroutines are simulated by recursive function calls. Any coroutine can call upon a primitive operation to request a tuple from a child : it can then transfer a tuple back to its parent through a return statement. The pipe structure preserves static information for each coroutine while it is inactive.

The coroutine at the base of the pipe is the first to be invoked. This coroutine requests a tuple from its child. The child in turn requests a tuple from its child, and so on down the pipe. Eventually, a coroutine at the other end of the pipe is called upon to deliver a tuple. This coroutine activates the paging mechanism and retrieves a single tuple from disc. The tuple is then passed back to its parent.

Once a coroutine has obtained a tuple from its child, it applies whatever operation is necessary. For instance, the coroutine corresponding to a projection transforms the fields within the tuple. It then returns the transformed tuple to its parent. A selection coroutine keeps requesting tuples from its child until finding one that matches the specified constraint. The tuple is then handed to the parent coroutine. The tuple is thus successively transformed and manipulated until it reaches the base of the pipe-line.

The coroutine at the base is a data sink : it discards the tuple and immediately requests another from its child. Thus the pipe is kept in motion until there are no more tuples to be processed. The overall effect can be viewed as a flow of tuples down the pipe, while requests for tuples travel up the pipe.

4.4. Listing And Assignment Of Relational Expressions.

Just as for the other relational operations, a coroutine is used to print tuples. This coroutine requests a tuple from its child and waits. When it receives a tuple, it prints it and then hands the tuple to its parent. A similar coroutine is used to assign tuples to an output relation. This method allows one to preserve an intermediate result of the pipe-line if so desired.

4.5. Implementation Of The Join, Intersection And Difference Operators.

The join, intersection and difference operator are implemented by having the pipe-line sort all received tuples into order before they are passed to the appropriate coroutines. It is then trivial to merge the sorted tuples. It has been shown in [BLAS77] that such a strategy is generally efficient.

During construction of the pipe, two sort coroutines are inserted between the operator node and its children. When activated, a sort coroutine reads all available tuples from its child, sorts them and places the result into a temporary file. It then hands back one tuple at a time to the operator coroutine. Figure 5 below shows the formation of a pipe-line corresponding to a listing of a join of two source relations.

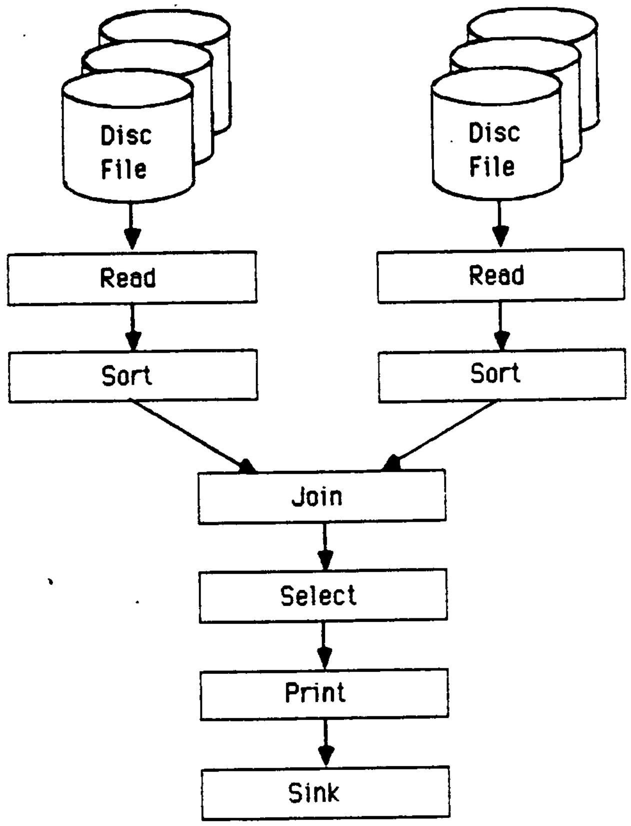


Fig. 5. This pipe-line structure corresponds to a join of two source relations followed by a subsequent listing of the result of the join. The join coroutine must receive tuples in sorted order. Therefore, sort coroutines are inserted into the pipe.

The same sort package used to reformat a relation may also be instructed to read tuples from a coroutine. Although sorting becomes a bottle-neck for processing these operators, it is generally preferable to locate matching tuples through a random access search or by making multiple passes over the data.

The fact that the sort package places its output into a real file aids the implementation of the join operator. The join coroutine may need to rescan one of its operands. It is easy to back up the scan when reading from a real relation file : it would be more difficult if the coroutine were reading directly from a child. If a rescan is necessary, it is likely that the tuples are still available in the buffer pool.

4.6. Advantages Of The Pipe-Line Algorithm.

The pipe-line algorithm allows an uninterrupted flow of tuples between successive processing steps. Generally, it avoids intermediate results having to be written to temporary disc files.

Another advantage is the ease with which algorithms can be coded to perform the various relational operations. The coroutine procedures themselves are small and efficient and each performs a well defined unit of work. The coroutines are written so that they are independent of one another. A coroutine requests a tuple via a primitive operation. No coroutine need know from where that tuple has come or to which parent it will be returned.

Generally, the coroutines transfer pointers to tuple buffers between themselves. Only rarely must an entire tuple be copied. The cost of activating a coroutine is small and so this algorithm requires little overhead. Indeed, it has been shown in [STOR77] that the cost of passing control between pipe-line nodes is not significant compared with other costs of a DBMS.

The join, difference and intersection coroutines each need to receive tuples in sorted order. Sorting, therefore, becomes a bottle-neck throughout processing. However, the bottle-neck is localised : a highly efficient sort package makes considerable difference to the overall performance. Profiling indicates that a third of the time taken to process an "average" query is spent in sorting. Coding part of the sort routine in assembler language would probably prove beneficial to the system's efficiency.

Because the coroutines are independent, there is no fundamental limit on the complexity of the pipe structure. Such a limit is instead imposed by the operating system. The number of source relations cannot exceed the permitted maximum number of open files. A complicated relational expression may, therefore, have to be split up into smaller processing steps. In order for this algorithm to function, there must be sufficient main storage to hold the pipe-line and the coroutine procedures. Space for holding page buffers and for sorting is also required.

5. Further Improvements

Sorting is a bottle-neck and, although the sort package is reasonably fast, the db program should recognise if it can be avoided. Most coroutines can be written so that they preserve the order of tuples flowing in the pipe. Db should optimise construction of the pipe to take advantage of this fact. It should eliminate and combine sorts wherever possible, as discussed in [HALL76] and [SMIT75].

The ability to update a relation file simultaneously by several users would also enhance the db++ system's usefulness. We feel that the major problem is perhaps one of semantics rather than implementation. What should be the correct action if an updating process crashes ? In other words, how should a multi-user update capability interact with the existing transaction and undo facilities [4] ?

[4] It is, however, already possible for simultaneous updates to be made to a database by several users working at the C programming level. For instance, the pharmacy application mentioned above comprises a server process that performs the actual changes to the database, and several

A third area of improvement concerns the db++ query language. Although this has proven itself to be useful for formulating complicated relational queries, it could perhaps borrow some ideas from conventional programming languages such as Pascal or Modula-2. For instance, a widely held point of view among software engineers is that a programming language should support as much static type-checking as possible, so that a compiler or interpreter can detect inadvertent errors before a program begins execution. These ideas would also benefit the db++ query language. It should support derived types such as sub-range types, enumeration types and set types.

6. History And Acknowledgements

The db++ programs belong to a family of related DBMS systems. In particular, db++ is related to the CODD system [KING83] which pioneered the use of coroutines to transfer control between pipe-line nodes. Whereas the db program uses recursive activations of procedures to simulate a coroutine mechanism, CODD uses a genuine coroutine implementation with each active coroutine possessing its own stack. CODD thus permits a generalisation of the pipe-line structure unobtainable with db++ : the pipe-line need not be a tree structure, but can be a directed graph. However, this additional flexibility is obtained, perhaps, at the cost of portability. CODD is only implemented on those systems where the coroutine mechanism described by Moody and Richards [MOOD80] exists. Furthermore, CODD is written in BCPL and is not oriented to the Unix programming environment.

In turn, CODD was developed from an earlier system called PRTV [TODD76]. All three systems share a common form of query language and all three use the technique of pipe-lining to evaluate database queries.

Work began on a primitive version of the db programs at the International Institute for Applied Systems Analysis, A-2361 Laxenburg, Austria. Concept ASA have built on this and the ideas of the CODD and PRTV systems to produce the db++ system. The db++ system includes enhanced functionality, has been improved in terms of processing speed and has now been ported to most flavours of Unix running on a wide variety of machines.

7. Conclusion

The db++ system is being used to maintain many diverse databases, each with its own special requirements. The choice of the relational algebra as the basis for the query language allows one to formulate both simple and complex queries. This fact, coupled with the db++ system's strong integration into Unix has facilitated construction and use of these databases. In particular, we feel that the lack of a specialised "db++ environment" has greatly helped db++ users in these tasks.

We have also shown how the B+-tree data structure can be coupled with a bit-map algorithm to yield the best of both : all the advantages of the B+-tree data structure together with a straightforward yet powerful transaction and undo facility. The implementation of the B+-tree and transaction algorithms is not difficult and has yet produced a storage method with clear, well defined semantics combined with the low storage and processing overheads associated with B+-trees.

We have also shown how the pipe-line algorithm aids the db++ system to process complicated queries. Besides avoiding unnecessary storage of intermediate results on temporary disc files, the pipe-line algorithm simplifies the task of coding algorithms to implement the various relational operators.

Any data processing system can be ultimately judged by how well it can be applied to solving real problems. We claim that the db++ system has been largely

client processes that make requests to the server.

successful in this respect.

8. References

ASTR76 *Astrahan, M. M. et al.,* - System R : A Relational Approach To Database Management - ACM TODS, Vol. 1, No. 2, pp. 97-137, (June 1976).

BLAS77 *Blasgen, M. W. and Eswaran, K. P.,* - Storage And Access In Relational Databases - IBM Systems Journal, No. 4, pp. 363-377, (December 1977).

CODD72 *Codd, E. F.,* - Relational Completeness Of Database Sublanguages - Database Systems, Courant Computer Science Symposia Series, Vol. 6., Prentice Hall, (1972).

CODD79 *Codd, E. F.,* - Extending The Relational Database Model To Capture More Meaning - ACM TODS 4, No. 4, (December 1979).

CONC85 *Concept ASA,* - Users' Guide To The DB Relational Database Management System - Wolfsgangstrasse 6, D-6000 Frankfurt am Main 1, (1985).

HALL76 *Hall, P. A. V.,* - Optimisation Of Single Expressions In A Relational Database System - IBM Journal Of Research And Development, Vol. 20, No. 3, pp. 244-257, (1976).

HITC77 *Hitchcock, P.,* - User Extensions To The Peterlee Relational Test Vehicle - Proceedings Of The 2nd International Conference On Very Large Databases, pp. 169-180 : North-Holland, (1977).

JOHN75 *Johnson, S.,* - YACC - Yet Another Compiler-Compiler - Computer Science Technical Report No. 32, Bell Telephone Laboratories, New Jersey, (July 1975).

KING79 *King, T. J.,* - The Design Of A Relational Database Management System For Historical Records - Doctoral Dissertation, Corpus Christi College, University Of Cambridge, Cambridge, England, (November 1979).

KING83 *King, T. J.,* - The Design And Implementation Of Codd - Software, Practice And Experience, Vol. 13, pp. 67-78, (1983).

KNUT73 *Knuth, D.,* - The Art Of Computer Programming, Vol 3., pp. 473-479 - Addison-Wesley, (1973).

KNUT77 *Knuth, D.,* - The Art Of Computer Programming, Second Edition, Vol 1., pp. 190-196 - Addison-Wesley, (1977).

MOOD80 *Moody, K. and Richards, M.,* - A Coroutine Mechanism For BCPL - Software, Practice And Experience, Vol. 10, pp. 765-771, (1980).

SMIT75 *Smith, J. M. and Chang, P. Y. T.,* - Optimising The Performance Of A Relational Algebra Database Interface - CACM, Vol. 18, No. 10, pp. 568-579, (October 1975).

STON75 *Stonebraker, M.,* - Getting Started In INGRES - A Tutorial - Electronics Research Laboratory Memorandum ERL-M518, University Of California, Berkeley, (April 1975).

STON80 *Stonebraker, M.,* - Retrospection On A Database System - ACM TODS, Vol. 5, No. 2, pp. 225-240, (June 1980).

STOR77 *Storey, R. A. and Todd, S. J. P.,* - Performance Analysis Of Large Systems - Software, Practice And Experience, Vol. 7, No. 3, pp. 363-369, (June 1977).

TODD76 *Todd, S. J. P.,* - The Peterlee Relational Test Vehicle - A System Overview - IBM Systems Journal, Vol. 15, No. 4. pp. 285-308, (1976).

YORK85 *University of York,* - Private Communication - unpublished, (1985).

FLORENTINE INVENTORS OF MODERN ALPHABETS

Chuck Bigelow

Stanford University

Bitmap screen displays and laser printers have enriched the appearance of computer literacy. Procrustean limitations of mono-case and mono-space that formerly degraded computer-produced text have been abolished. We now can enjoy the luxury of reading and printing text in lower-case as well as in capitals, in italic and bold styles as well as in roman, in proportionally-spaced fonts of different sizes as well as in monospaced fonts of a single size, in justified as well as in ragged-right columns.

Curious users and designers of typographic displays and printers may wonder where the shapes and styles of our alphabet come from. The digital letters of modern computer systems inherit their shapes from traditional analog letter forms. These shapes are arbitrary and abstract, but they were designed by scribes seeking legibility for the reader and productivity for the writer — goals that concern us today.

Many of the features of our modern alphabets were developed during the Renaissance by a small but influential group of Humanist scribes and scholars who worked in the city of Florence. It is to their memory that I would like to offer this tribute. Although historical rather than scientific in content, this brief account of how a writing system developed by a few inspired visionaries became an international standard may perhaps sound familiar to user of UNIX systems.

In the middle and late 14th century, the poet Francesco Petrarca (whom we call Petrarch in English) and Coluccio Salutati, a scholar who was the chancellor of Florence, became concerned about legibility of letterforms. In their thirst for knowledge, they read more than their predecessors and thereby suffered more from the problems of deciphering poorly written texts in the "gothic" or "black-letter" scripts. They preferred early medieval manuscripts written in an open, legible script developed at the court of Charlemagne and used from the 8th through the 12th centuries. The Humanists called this Carolingian script the "littera antiqua" (or "antica") because it had been used for the works of classical authors.

Coluccio Salutati saw that this script could be the basis for a new alphabet, better adapted to the needs of modern literacy. Around the turn of the 15th century, Coluccio hired a young scribe named Poggio Bracciolini who had studied to be a notary at the Florentine Studio. With Coluccio's encouragement, Poggio became an expert copyist of the "littera antica", and began to produce new manuscripts in this revived style around 1402-03. The original Carolingian script was essentially a mono-case alphabet. Poggio studied inscriptional capitals from the early Roman Empire, and integrated pen-written versions of these majuscule (capital) forms with his minuscule (lower-case) script. He perfected his "duplex" alphabet by 1408, and it was quickly imitated by his contemporaries.

Descendants of Poggio's "humanist bookhand" became the models for roman printing types developed by Nicolas Jenson and Francesco Griffo later in the century. These early analog types are the direct ancestors of the digital types we use today. Poggio also worked on methods of justification, experimenting with unorthodox word divisions as well as the use of abbreviations, and may have influenced the attitude toward justification held by Italian printers later in the 15th century. Poggio went on to become secretary to the Pope and eventually chancellor of Florence.

Coluccio and Poggio had a fellow Humanist named Niccolo Niccoli. Unlike Poggio, who was a professional scribe, Niccoli copied manuscripts for his own scholarly purposes. Rather than beauty, legibility and speed were his chief aims. Niccoli's script was, like Poggio's, a revival of the Carolingian script, but it was a running hand: slanted, with frequent joins between letters. Niccoli developed his cursive or "corsiva" in contrast to the formal or "formata" style of Poggio. Niccoli encouraged other scribes to abandon the gothic scripts and use his and Poggio's newer and more legible writing styles.

In time, Niccoli's informal "corsiva" became a more formalized, elegant style that was first cut in printing type in 1501 by Francesco Griffo, for the Venetian printer Aldus Manutius. The Aldine cursive inspired the development of our modern italic type forms.

Other Florentine scribes adopted and developed these formal and cursive scripts, producing the canonical styles that we call in English "roman" and "italic". Among them were Giovanni Aretino, Antonio di Mario, Gherardo del Ciriagio, and Antonio Sinibaldi. After the invention of printing and its migration to Italy, innovation in letterforms passed to the city of Venice, where the Florentine scripts were reinterpreted in a new technology.

Today, traditional printing types are again being reinterpreted in the still newer technology of digital typography, but our modern goals are the same as those of the Florentine scribes, legibility for the reader, speed for the writer, and, in reflective moments, an answer to the question of how ideas can be clearly expressed in these abstract forms that, as in Plato's metaphor, flicker before us like shadows on the wall of a cave.

REFERENCES

[These remarks are based on {The Origin and Development of Humanistic Script} by B. L. Ullman, Edizioni di Storia e Letteratura, Rome, 1960.]

M. Bologna, C. Romoli - OLIVETTI GSRI/DMR Pise, Italy

AN ENTITY RELATIONSHIP DATABASE FOR SOFTWARE ENGINEERING: THE PORTABLE COMMON TOOL ENVIRONMENT APPROACH

ABSTRACT

The Portable Common Tool Environment (PCTE) project is carried out inside the ESPRIT programme of the Commission of the European Community.

The project aims at the definition and the implementation of a common framework, within which various software tools can be developed and integrated in order to provide a complete environment for software engineering.

The two main goals of the project are portability across a wide range of machines and compatibility for assuring a smooth transition from existing software development practices.

The technical approach to portability is discussed in this paper: UNIX* is the means by which PCTE will be widely available; the PCTE basic mechanisms are built on top of UNIX: in fact PCTE can be seen as an extension of UNIX in the area of distribution, friendly user interfaces and database for software engineering.

This last aspect is described in this paper: functional as well as implementation aspects will be dealt with.

* UNIX is a trademark of Bell Laboratories

1. THE PORTABLE COMMON TOOL ENVIRONMENT PROJECT

The project "A Basis for a Portable Common Tool Environment" (PCTE) is carried out by a consortium composed by Bull (France), GEC and ICL (United Kingdom), Nixdorf and Siemens (Federal Republic of Germany) and Olivetti (Italy) in the context of the European Strategic Programme for Research and Development in Information Technology (ESPRIT).

The project started at the end of 1983 is now in the implementation phase (part of the results are in fact already available) and will run until September 1987.

The main goal of the project, which is included in the Software Technology area, is to define, design and implement a software system which

- can be the basis for the development of a complete Software Engineering Environment;
- is as much portable as possible;
- is generally powerful enough to guarantee future enhancements;
- is consistent and compatible with the perceived major trends in the European software industry.

In addition, the project aims to achieve a wide spread availability of the system throughout the ESPRIT community both in a short time frame and in a cost effective manner.

Due to its portability and widespread availability the UNIX system has been chosen as the basis for PCTE: PCTE can in fact be seen as an extension of UNIX in the area of distribution, friendly user interfaces and database for software engineering environments.

Two different complementary approaches have been identified for the implementation of PCTE basic mechanisms:

- the so called 'add-on kernel' approach which is based on the UNIX V kernel plus extensions and a controlled set of modifications;
- the so called 'Black Box' approach which is again based on the UNIX V kernel but:
  - uses the UNIX V kernel as a 'Black Box' (i.e. no changes, no modifications);
  - it has UNIX V as primary objective but it is not restricted to it;
  - it offers Ada* interfaces in order to have a multilanguage environment.

This approach is discussed in this paper: after a description of the basic PCTE facilities, the software engineering database is described in detail together with implementation aspects.

* Ada is a trademark of the Ada Joint Program Office

Part of this paper has been derived from PCTE documents namely PCTE Functional Specifications rel. 1.3 (PCTE 85a) and papers presented at the 1985 ESPRIT Technical Week in Brussels (PCTE 85b) and (PCTE 85c).

2. OVERVIEW OF THE BASIC PCTE FUNCTIONALITIES

PCTE has been designed mainly to serve as a basis for the construction of a Software Engineering Environment; the three main aspects of PCTE are:

- Basic Mechanisms;
- User Interface;
- Distribution.

These aspects will be briefly described in the remaining part of this section.

2.1. Basic Mechanisms

The basic mechanisms are defined as a set of programs callable primitives; they mainly cover three aspects: program execution, communication and manipulation of the various 'objects' existing in the environment.

They can be subdivided into five categories, namely Execution, Communications, InterProcess Communications, Activities and Object Management System.

Execution primitives deal with the notion of a program in execution: they define how a program can be started or terminated, how it can be controlled and so on; these mechanisms are very similar to the UNIX ones.

Communication primitives allow program to access the file type unstructured data (a UNIX file implementing the notion of 'contents of an OMS object' (see later): these mechanisms are the conventional input-output facilities of UNIX.

Mechanisms like Shared Memory, Signals and Pipe are provided for processes to communicate: these mechanisms are also strictly related to the corresponding UNIX ones.

Activities have been introduced to cover the lack of data access synchronisation and recovery mechanisms of UNIX.
The PCTE activities mechanisms implement the concept of transaction in Software Engineering Environments that have been adapted.

Object Management System is the subject of the following section.

2.2. User Interface

An object oriented User Interface allow the user to view multiple sources of information in a uniform way: this implies that several processes may run in parallel on the same workstation.

User Interface mechanisms establishe the communication between the user and the various PCTE processes providing window management and input/output control.

2.3. Distribution

The PCTE preferred architecture is a Local Area Network of powerful single-user workstation.

The distribution mechanisms provides:

- transparent distribution of functions of the OMS, process execution, process structure and inter process communications;
- network administration (control of the OSI transport layer);
- distribution management (control and configuration).

3. OBJECT MANAGEMENT SYSTEM

A software engineering environment has to manage a certain number of entities designated by the generic name 'object', a term widely used in the area of software engineering environments (Dod 80).
An object can be a file in the traditional sense, a program, a product designed for a given target machine, a document, a structured object like a library of software components or something more abstract like a project.

The goal of the Object Management System (OMS) is to manage all these objects in an uniform way, i.e. to provide storing mechanisms, convenient access means and to manage their various properties and interrelationships, in particular the various dependencies that must be kept in order to support the different versions of a software product, to control the modifications applied to a given object and to know their consequences.

The objects base, or "database" is managed by a set of operations which constitute the Object Management System.
The OMS can be seen as an evolution of the traditional File Management System, replacing the predefined structure of the file systems (e.g. the hierarchic structure of UNIX) by a structure which can be adapted to the needs of different environments. It is in fact a specialized Data Base Management System, characterised by its adequacy to the above mentioned types of objects.

It includes some facilities which are not provided in general-purpose DBMS but implies also some limitations: for example the lower limit of the object's granularity may be a little higher than in traditional DBMS.

An other important characteristic is the fact that any access to an object is the result of a navigational request or pathname whose syntax is a superset of the UNIX one. Furthermore, the database is distributed in a transparent way.

A model is proposed to define the various types of objects, properties and relationships which constitute the structure of the database.
This model is in the "entity-relationship" family (Chen 76). It can be also seen as an extension of the network model, supporting several kinds of relationships and subtyping facilities.

A schema, in traditional DBMSs, is a set of type definitions describing the structural and semantic properties of the database.
A database is a collection of instances of type definitions. The OMS database is the integrated, central repository of the environment information for all projects, users and tools of a PCTE installation. The schema is a means of integrating tools around commonly accessed data structures.

However, projects, users and tools always access the portion of the information they are concerned with according to their specific Working Schema.

A Working Schema is a partial view over the overall schema. As traditional "external schemas" (also sub-schemas or views), Working Schemas are filters which shield database accesses from the actual database structure.

As a matter of fact, Working Schemas allow the coexistence of several, otherwise conflicting views of the database. Working Schemas play a paramount role in supporting:

- portability of tools and tool sets among projects and among PCTE installations

- integration of new tools around existing data structures

- project and user level definitions of database structures, which in turn can support, for instance, specific methodologies and working styles.

  Flexibility augments together with data|=_application independency.

Schemas are evolving entities and this is especially true in Software Engineering Environments. OMS supports incremental schema modifications, which can take place in parallel with database exploitation (so no "technical shutdown" is needed because of schema changes).

The type definitions making up the overall OMS schemas are organized into a collection of (small) sets of

definitions called Schema Definition Sets (SDS).
Each SDS is an OMS object and holds the description of a partial view over the overall OMS schema. Modifications to the schema are always carried out as SDS updatings. The mappings between SDSs and the actual schema are automatically maintained by the OMS schema management facilities.

Working Schemas are made up of one or more SDS in the same way executable programs are made up of one or more compilation units.

Through SDSs, OMS supports the possibility of decentralising at project and user levels the management of schema definitions: there is no need for a centralised "Schema Administrator" authority.

The integrity of the database is enforced first by ensuring that the manipulation operations respect the semantic properties declared in the Working Schema and second by controlling their consistency in the context of concurrent Activities.

One can distinguish two aspects of OMS functionalities: the description of the schema and the manipulation of the database.

3.1. Object

An "object" is an entity which can be distinctly identified. A specific source file, tool or program library is an example of an object.

Objects are classified into different "object types" such as "source_file", "tool", "program_library" (Smi 77). All objects of a given object type have some common characteristics:

- a set of properties, called attributes qualifying the object.

- a set of links starting from the object.

- optionally, the contents

3.2. Object Attribute and Link Attribute

An object attribute defines an intrinsic property of an object. Typically, a creation date is an object attribute.

A link attribute qualifies a link and indirectly the object which is the destination of this link. Let's consider for example, the links defining the set of compilation units which constitute a program. Each link could be qualified by information provided by the linker (e.g. code address of the compilation unit in the resulting load module): the value of a link attribute depends both on the destination and on the origin of the link.

3.3. Link

A link is a unidirectional association between one origin object and one destination object.

A link can be used to create a simple reference from an object to an other one, or to modelize structures like the directory one, represented by an object from which several links are leading to component objects.

A link can have arity "one" or "many":
in the first case only one link of this type can start from an object; if the arity is "many" several links of this type can start from the given object. In this case , a key must be used in order to distinguish the different links.
The link is also the basic element of a pathname.

A link type can be declared as a member of a relationship type or individually.

3.4. Relationship

A relationship is a pair of two links such that the origin of each is the destination of the other. According to the arity of the two link types, the terms "one-one", "one-many" and "many-many" can be used for a relationship type.

3.5. Pathname

A pathname is a sequence of link names starting from a reference object associated to the context.

4. IMPLEMENTATION OF THE OBJECT MANAGEMENT SYSTEM

This section deals with the implementation aspect of OMS in the Black Box approach: remember that by Black Box approach we mean a PCTE implementation which does not make any modifications to the UNIX kernel.

The main goal of the Black Box approach is to make available PCTE functionalities on a wide range of host computer and environment; giving the major emphasis on the portability across different versions of UNIX.

The key issue for the success of PCTE is its wide and easy availability; since the proposed implementation is UNIX based, portability should be easily achieved: in order to make the system fully portable modifications to the kernel should be avoided.

In this respect, the cost of porting PCTE can be really zero if a subset 'standard' of UNIX primitives is used. From the beginning of the developments we took into account the UniForum committee standard and today the SVID definition (XOG 85) constitutes our reference manual.

We are conscious that in some areas of PCTE this choice could contrast with efficiency; on the other hand we have to consider that several organizations will use PCTE on their systems, which could run different versions of UNIX.

The PCTE/OMS environment

The PCTE/OMS is the first result of the Black Box implementation.

It has been developed using highly system independent programming techniques (as suggested in (XOG 85)) and a subset of UNIX system calls.

Up to now it has been installed on a wide range of machines and UNIX (and UNIX like) operating systems.

In the PCTE/OMS environment it is possible to define several PCTE/OMS subsystems which are independent of each another.

In fact, a PCTE/OMS subsystem can be defined as a complete self-contained PCTE environment in which a number of SDS and volumes can be created.

More than one process can run simultaneously on the same subsystem and access to the objects and Schema Definition Sets defined in that PCTE/OMS subsystem.

- Volumes representation

Each volume is represented under the unix file system as a directory which contains several subdirectories: the 'TKx' directories which contain the files representing the objects

and a counter file which allows the internal numeration of the objects stored in that volume.

The system volume also contains an 'SDS' file with the representation of the system schema.

- Object representation

Each OMS object is represented by a file in the database directory.

This file contains the attributes of the object, the links starting from that object and the attributes of the previous links; a contents attribute is represented by the path name of the file holding the contents.

In this way a contents attribute may be considered as a normal unix file.

The unix name of the file representing the object is the ascii representation of the internal counter plus the suffix '_0'; for the file representing the contents of the object the name is the same internal number plus the suffix '_F'.

5. CONCLUSIONS

An implementation of the OMS interfaces is the first result of the Black Box approach, which could run separately from the entirely PCTE environment.

That OMS "library" has been successfully installed on several host machines (Olivetti M28, 3B2/X, Digital VAX, APOLLO, SUN) running different versions of the UNIX operating system (SYS V, XENIX, SYS III, Version 7, BSD4.1 & BSD4.2, ULTRIX, XENIX).

These successfull installations are the verification of our approach to portability.
A mono_user version has also been installed on M24 and IBM/PC running the MS/DOS operating system with XENIX libraries.

Tools for managing the installation and for using OMS basic functionalities have been produced.

To analyze and exploit the power of the OMS model some tools have been realized on top of the OMS library: an example of which is the PMAIL mailing system.
PMAIL (PCTE MAIL) is a program allowing the storage and retrieval of UNIX mail into the OMS.

It is possible, using that tool, to retrieve the message of the mail from Unix and to catalogue it in the private OMS database.

While cataloguing a mail the user is able to provide 'keywords', for easy and fast retrieval of the mail.

The main advantages of the PMAIL tool, derived from the use of the Object Management System interfaces, are:

- privacy: mails are stored in a private (or in a private part) OMS;

- fast retrieval: providing 'keywords' related to the contents of the mail allows easy and fast retrieval of mail messages;

- flexibility: keywords are not predefined; each user can define his/her own keywords.

- reliability: the OMS internal mechanisms maintain the consistency and integrity of data.

Following the example of the mailing system it is possible to define and develop similar tools to enhance the Unix tool set.

6. REFERENCES

(DOD 80) Department of Defence Requirement for Ada Programming Support Environment "STONEMAN", February 80

(CHEN 76) Chen P.P., The Entity-Relationship Model: Toward a Unified View of Data; ACM Trans. on Database Systems, 1, 1976

(PCTE 85a) Bull, GEC, ICL, Nixdorf, Olivetti, Siemens "PCTE: A basis for a Portable Common Tool Environment. Functional Specification", Third Edition, 1985

(PCTE 85b) Bull, GEC, ICL, Nixdorf, Olivetti, Siemens "PCTE: A basis for a Portable Common Tool Environment. Design guideline", Paper presented at the ESPRIT Tech. Week, Brussels, 1985

(PCTE 85c) Bull, GEC, ICL, Nixdorf, Olivetti, Siemens "Overview of PCTE: A basis for a Portable Common Tool Environment.", Paper presented at the ESPRIT Tech. Week, Brussels, 1985

(SMI 77) Smith J.M., Smith D.C.P. "Database Abstraction: Aggregation and Generalization", ACM TODS Vol.2 June 1977

(XOG 85) XOPEN Portability Guide, July 1985

SIGMINI

AN INFORMATION SYSTEM WITH A DYNAMIC ARBORESCENT STRUCTURE, INCORPORATING A SELF STRUCTURING MODEL

Charline Brisbois, Union Minière, Belgium
Patrick Mordini, Ecole Nationale Supérieure des Mines de Paris
CAI, France

INTRODUCTION

The software Sigmini is designated to treat heterogeneous informations, including quantities, which have complex interrelations. It is related to both databank systems and classical documentary retrieval systems. **The basic idea is to avoid having to declare in advance either the data or their relations.** The "self structuring model" is managed by handling (element = value) couples hierarchically structured by parentheses ; the element being more generic and the value more specific. The system allows the user to store the data and the desired relations between them, at the time the data are acquired without referring to a predetermined scheme. The search of a bank where the data are stored according to this kind of model has to contain an explicit interrogation of the structure of the data beside the usual boolean operators. First, a classical interrogation is effected, using an inverted file i.e. without reference to a structure, to make a preselection; an interrogation with structures on the preselected data is then made and this, in a very simple language.

Sigmini is an online retrieval system designed for minicomputers using the Unix operating system (system V). Sigmini has been developped jointly by a Research Center of the Ecole des Mines de Paris and Union Minière. It has specifically been conceived to be portable. The programs are written in Fortran. Except for the input-output routines written in C, the software can be easily adapted to other systems.

The existing Sigmini databanks cover the following fields :

- economical and technical informations concerning non-ferrous metals;
- inventory and description of archaeological patrimony.

The examples used hereafter have been taken in one of the banks constituted by Union Minière, using the Sigmini system.

1. INPUT of DATA

1.1 The elementary information

Every basic elementary information is introduced by a couple (element=value). The element gives a generic information : country, company, period of time, grade, title. The value gives a specific information for a given element.

There are several types of elements :

- semantic type	i.e. Country = Belgium
- standard type	Company = Amax
- numerical type	Registration number = 312523
- numerical type with 2 bounds	Grade = 3 Period of time = 800101 A 861231
- commentary type	Title = Process for separating and recovering Ni and Co.

There are data of qualitative type and data of quantitative type. They can be classified in two categories, one with strong information as the semantic and the standard and one with weak or fuzzy information as the numeric and the commentary.

All the elements are stored in a dictionary. The values of the elements of the semantic and standard types are also stored in a dictionary.

For a semantic type, i.e. a Country, one can determine the relations existing between regions, as well as their belonging to an economical entity, i.e. the EEC, or a continent, i.e. North America.

For a standard type, i.e. a Company, the values are stored which moreover avoid confusion and spelling mistakes. But these values cannot be grouped together to form semantic entities.

The values of the other types of elements are only stored in the database.

For the numerical type, the value can be introduced both as an integer or as a decimal number.

The numerical value with 2 bounds represents a couple of values determining the range. If only one numerical value is given, it means that the length of the range is zero.

The value of the commentary type is a chain of unspecified characters of a length smaller than 32.000 characters.

The different types of elements, and particularly the semantic type, are further explained in the chapter concerning the dictionary.

1.2 Relations between data

Every homogeneous group of information or record can be hierarchically structured by introduction of parentheses. The number of levels of the hierarchy is defined at the system generation.

Example

The record given below describes the evaluation of a Au/Cu deposit and the project of construction and exploitation of a mine in Papouasia. This record also contains the construction costs, the production objectives as well as the percentage of the participations taken by various compagnies. This example shows that the data and the structure are given simultaneously. Every element can be placed everywhere, and can be repeated if necessary.

```
REGISTRATION NUMBER = 310928
YEAR = EC080
MICROFILM NUMBER = 8040527
JOURNAL =
DATE = 800718
REFERENCE =
LANGUAGE = ENGLISH
TYPE OF DOCUMENT = ARTICLE
VALUE OF DOCUMENT = 3
CONTENT OF ANALYSIS = 2
   (GOVERNMENT = PAPOUASIA NEW GUINEA
      (COUNTRY = PAPOUASIA NEW GUINEA
       LOCATION = STAR MOUNTAIN
         (DEPOSIT = EVALUATION,DRILLING
            (MATERIAL = ORE
               (ELEMENT = Au/Cu
                RESERVES = 410000000
                UNIT OF MASS = T
               (ELEMENT = AU
                GRADE = 3 A 3.5
                PERCENTAGE = G:T
         (MINE = CONSTRUCTION/PROJECT
          NAME OF MINE = OK TEDI
            (MATERIAL = ORE
               (ELEMENT = AU/CU
                COST = 800000000
                MONETARY UNIT = US DOLLAR
         (MINE = EXPLOITATION/PROJECT
          NAME OF MINE = OK TEDI
            (MATERIAL = ORE
          (ELEMENT = AU/CU
           PRODUCTION = 12000
           UNIT OF MASS = T/J
(COMPANY = FLUOR AND METALS INC
STATUS = ENGINEERING,TECHNICAL ASSISTANCE
(COMPANY = DANCO
STATUS = ASSOCIATION
PARTICIPATION % = 37.5
(COMPANY = MOUNI FUBILAN DEVELOPMENT COMPANY
STATUS = ASSOCIATION
PARTICIPATION % = 37.5
(COMPANY = METALLGESELLSCHAFT A.G./SIEMENS A.G./
       KABEL-UND METALLWERKE GUTEHAFFNUNGSHUTTE
       KABELWERK BERLIN GMBH
STATUS = ASSOCIATION
```

1.3 Possibility to adapt and fill in the gaps in the structure of the data

If one wants to update the record on the Au/Cu deposit in Papouasia, i.e. having obtained additional information about the copper grades, the gold grade being the only known at the beginning, one will have to modify the structure of the record. The gold and copper grades will have to be placed on different nodes, not to mix up the grades of gold and copper.

This shows that the structure is dynamic and can be adapted in function of the information evolution, as well concerning additional structures as changing the values of the elements.

1.4 Efficient use of the self-structuring model

Retrieval of the data by a simple and adequate way in a "self structuring model", implies to make the analysis of the information on pre-established scheme and structuring rules, the more that the system allows empty nodes on one or more levels. These rules are unknown by the system.

The scheme and the structuration can be preestablished as "check-lists" in function of skeletons for the different kinds of records. These skeletons contain the possible elements of the records and their structure. These "check-lists" are external to the system and can be modified in function of the needs. A "check-list" can be a simple form, a conversationnal "question-answer" menu or an editor dedicated to the Sigmini data acquisition. This kind of editor is already used with the Sigmini system.

2. DICTIONARY

The dictionary has 2 objectives :

- to check the elements names and the standard and semantic values;
- to create names codes to store them in the database.

a. Content of the dictionary

The dictionary includes several fields :
- the element field;
- the values fields attributed to standard and semantic types elements.

Numerical and commentary types are characterized by the fact that respectively any numerical value or any character block can be attributed to an element. These values are not stored in the dictionary, but the elements are.

In standard type, the elements are connected to a well defined list of possible values. The standard data acquisition is done with closed dictionary so as to exclude terms wrongly spelt or not yet accepted in the dictionary. The dictionary can be opened so as to introduce new elements or new values attributed to the elements. This can be done during data acquisition or by a specific operation.

Two kinds of information are stored in the dictionary, on one hand the elements and on the other hand the various groups of values attributed to elements of the standard and semantic type.

In each of these sets it is possible to link the terms by synonymical relations. So, the same value can be introduced in whole, shortened form or in foreign languages. One can use indifferently any one of the synonymes.

For instance Company = Union Minière
or
SOC = UM
or
Société = Union Minière

A special device allows, to identify to which language belongs a certain term, and as far as the different languages have been introduced, a record can be edited in another language than the one used for the introduction.

In principle, all the values of a standard element belong only to that element. However, a mechanism allows to make compatible all values of an element with all values of another element, which consists to group together the values of those elements.

For instance : Country, Country destination and Country origine are compatible elements and they have the same values.

Several databases can have the same dictionary ; it is possible to distinguish elements and values belonging to one or more databases.

b. Semantic dictionary

The semantic dictionary explains the meaning and the reciprocal relations of elements values of this type. It is more powerfull than most hierarchical thesaurus of documentary systems.

For instance : Country is an element of semantic type. If a record is introduced with the value Country = Québec, this record will be selected for a question about Québec but also for a question about Canada.

If one introduced a record with the value Country = Great Lakes, the record will be retrieved for a question about Canada or about the United States. This shows that the semantic dictionary can handle problems with relations between non-hierarchical values.

3. RETRIEVAL

The search of a bank using the Sigmini system is done in two phases. The first called preselection, is based on an inverted file. The second called selection, examines the content and the structure of every preselected record. For this second phase, the structure operators are used to describe the required relations between the preselected data.

3.1 Preselection

The inverted file allows a search, per element or per value, of records that contain the required elements or values.

All elements or all values do not have to be inverted ; the selection of the elements and the values to be inverted is decided at their introduction in the dictionary, only if they present a selective character.

Examples :

- Element Country is inverted on each of its values. During a search, the inverted file gives all the record identification numbers for the requested value of the element Country.

- Element Production is inverted as it is, without its values. All the records containing the element Production are given when requested.

Let's take a question as example : import of coal in West Germany from Australia.

This question can also be read as follows : export of coal from Australia to West Germany. In Sigmini language, it becomes :
1. (Country of Destination : ou : Country) = West Germany
2. :JU:
3. (Country of Origin : ou : Country) = Australia
4. :AS:
5. Market = import
6. :AS:
7. Material = (coal:ou:coke)

The lines 1, 3, 5, 7 form what is called the selecting criteria. The lines 2, 4, 6 contain the structure operators. A selecting criterion can be a combination of criteria and structure operators.

In the preselection phase, the structure operators of lines 2, 4, 6 are automatically replaced by the Boolean operator "ET".

The search can be made criterion per criterion so as to obtain at each step the number of preselected records.

For the example mentioned above, all the records containing West Germany as "Country or Country of Destination" And "Australia as Country or Country of Origin" And Import as Market And Coal or Coke as Material, are selected.

This preselection is stored in a working file containing the records that might suit if the desired structure is respected.

Records obtained during the preselection may seem too large or too small to the user. He can, of course, formulate again his question in a larger or more restricted way. If the preselection result seems acceptable, the selection phase is started.

In the example mentioned above, the record n° 4884 has been preselected.

```
REGISTRATION NUMBER = 310928
YEAR = EC080
MICROFILM NUMBER = 8140848 to 8140852
JOURNAL =
DATE = 810723
LANGUAGE = FRENCH
TYPE OF DOCUMENT = ARTICLE
DOCUMENTARY VALUE = 3
CONTENT OF ANALYSIS = 3
   (ORGANISM,INSTITUTION = EUROPEAN ECONOMICAL COMMUNITY
      (COUNTRY OF DESTINATION = BELGIUM/DANEMARK/
               FRANCE/GREECE/IRELAND/ITALY/LUXEMBURG/
               NETHERLANDS/UNITED KINGDOM
      (COUNTRY OF ORIGIN=UNITED STATES/CANADA/AUSTRALIA/
               SOUTH AFRICA/POLAND/U.S.S.R.
         (MARKET = TRADE/IMPORT
          PERIOD OF TIME = 790101 A 801231/ >810101
            (MATERIAL = COAL
               (QUANTITY =
      (COUNTRY = BELGIUM/DANEMARK/WEST GERMANY/
               FRANCE/GREECE/IRELAND/ITALY
               LUXEMBURG/NETHERLANDS/UNITED KINGDOM
         (MARKET = PRODUCTION/PERSPECTIVES
          PERIODE = >810101
            (MATERIAL = COAL/COKE
         (MARKET = TRADE/SALE/DELIVERY
          PERIOD OF TIME = 790101 A 801231/ > 810101
      (MATERIAL = COAL/COKE
   (MARKET = STOCK
      (MATERIAL = COAL/COKE
```

3.2 Selection

As mentioned already, the record n° 4884 was preselected. When examining this record, one notices that it gives no information about the coal import in Germany.

It contains data about the coal import from Australia but not destinated to Germany, but gives information about the production of coal of several countries, among which Germany.

The structure operators will avoid that kind of "noise" in selecting the records.

In this example, only two structure operators were used, the most frequently used are :

JU C1:JU:C2 means that the critera C1 and C2 are located on the same node of the structure.

AS C1:AS:C2 means that C1 and C2 have hierarchical links in the tree and that C1 comes before C2.

DE C2:DE:C1 means that C2 and C1 have hierarchical links in the tree and that C2 comes after C1.

FR C1:FR:C2 means that C1 and C2 have the same father.

PE C1:PE:C2 means that C1 is the father of C2.

All the operators have their negative equivalent.

For instance :

NJU C1:NJU:C2 means that C1 has to be located at a node on which C2 is not present

It is possible to forbid the presence of two critera at the same node of a record.

For instance :
:NON:(C1:JU:C2).

All element-value couples of the records can be obtained whithout having to mention them, in excluding the couples one does not want.

Example : to find all information about the economical activity of Zimbabwe, except for information on the market

(@ :NID:Market):FI:Country=Zimbabwe (NID means non identical).

The notation "@ " allows moreover :

- to describe one, and only one, element-value couple of a set, so as to impose to this element-value couple as many relations as wanted.
- to write questions "in loop".
- to describe relations of the "grandfather" type for which no specific structure operator is designed.

During the selection, Sigmini reads through all preselected records and examines if non inverted values which are required are present and if the structures of the record correspond to the structures asked in the question.

The search of a Sigmini bank is conversational. The user can adjust his question in function of the "noise" or the "silence" that he obtained both as preselection and selection phases. As soon as the first record is selectioned, it is displayed on the terminal. The user may at any moment decide to continue or to interrupt the selection, in case the answers are not sufficiently relevant or if the question has not been formulated correctly.

The system permits to store the formulation of each question so that it can be reused, i.e. for selective diffusion of information or SDI.

4. SIGMINI and UNIX

SIGMINI is presently available only on mainframes and not yet on microcomputers. The languages used are Fortran and C.

The input data are read on the "standard input" so that they can be easily obtained from a terminal, a disk file or a tape. The output data are put on the "standard output" so that they can be stored on a file or directed to the "standard input" of another program using the "pipe" mechanism. The messages for the user are written on the "standard error".

It is possible to pause a process, to run another program and to resume the previous one, i.e. during the search, one can check the spelling of a term in the dictionary and resume the search with this term. It is possible too to send a result of a searh to another user by the mail.

ANNEXE

BRIEF DESCRIPTION of the PRINCIPAL FUNCTIONS of SIGMINI

1. Input and update in the dictionary

Data input can be made :
- in a conversational mode;
- in batch processing, from a tape or a disk file.

The available commands allow to :
- create the elements and the values;
- update the synonymical relations;
- print the dictionary;
- establish inverting types of elements and the compatibility between values of some elements.

2. Input of the records in the database

The input can be done in different ways, from a terminal, a tape or a disk file. The system verifies each record ; the records with mistakes are rejected and have to be introduced again after corrections. An updating program allows to correct the rejected records without having to retype them entirely.

The conversational data acquisition is checked by a special program. The check-lists composed of skeletons of the different kinds of records are used to guide the data capture. The program checks the syntax and the conformity with the elements and the values in the dictionary.

3. Retrieval

The user has to indicate in advance which bank or part of a bank he wishes to browse.

Each user has one or more questionnaires at his disposal in which he can enter up to 42 different questions. In a questionnaire, every question has a reference number, which allows to use it again for a new question or a SDI.

There are three possibilities at the introduction of a question :

- syntaxical analysis only ;
- syntaxical analysis and preselection;
- syntaxical analysis, preselection and selection.

4. Display of the data

The preselected and selected records can be displaid on a screen or on a printer. It is possible to display parts of the records.

5. Example of retrieval

In the example below, the letters printed in bold are typed by the user. The comments are in italic. The rest are answers given by the computer.

```
 QUESTIONNAIRE?

$EQ  Display the names of questionnaires

        QUEST1:ECO1
        QUEST2:ECO2

        QUESTIONNAIRE?
$RQ/ECO2/       Call the questionnaire ECO2
        3:
$LQ     List all the questions of the current questionnaire
        1:(SOC=METALGESEL:JU:STATUS=ASSOC):JU:SOC=DEGUSSA/

        2:ELE=GA:DE:UTGE:IELEC/
        3:
$EC/2/  Display on screen the records selected by the question 2


      287 ****************************************
           REGISTRATION NUMBER=104133
           YEAR=ECO81
           NUMBER MICROFILM=8110197
           JOURNAL=
           DATE=801100
           REFERENCE=
           LANGUAGE=ENGLISH
           DOCUMENT TYPE=ARTICLE
           DOCUMENTARY VALUE=2
           CONTENT OF ANALYSIS=3
     ((    COUNTRY=UNITED STATES
             (MARKET=DEMAND/PERSPECTIVES
                FIELD OF GENERAL UTILIZATION=ELECTRONIC INDUSTRY
                FIELD OF SPECIFIC UTILIZATION=SOLAR CELLS/
                PHOTOVOLTAIC EFFECT
                PERIOD OF TIME=900101 A 991231
                 (MATERIAL=METAL
                      (ELEMENT=GA
                       QUANTITY=/

**STOP:1          CONTINUE:0

0   continue the display of the records
```

```
    826 *****************************************
        REGISTRATION NUMBER=104144
    (...................)

    3:
    SOC=COMINCO:AS:COUNTRY=CANADA:AS:(MINE:OR:CONCE)/
                                                   |
    QUES392   UNKNOWN VALUE
                                  question 3 : wrong

        3:
    SOC=COMINCO:AS:COUNTRY=CANADA:AS:(MINE:OR:CONCENTRATOR)/
                                       question 3 : preselection only
    NUMBER OF RECORDS PRESELECTED:15
    4:
$EX/3/  selection of the records preselected by question 3

    NUMBER OF RECORDS SELECTED:10
$EC/3/
    179 *****************************************
        REGISTRATION NUMBER=410442
        YEAR=EC081
        MICROFILM NUMBER=8110085
        JOURNAL=
        DATE=801216
        REFERENCE=
        LANGUAGE=ENGLISH
         DOCUMENT TYPE=ARTICLE
         DOCUMENTARY VALUE=3
         CONTENT OF ANALYSIS=2
         (   COMPANY=COMINCO LTD
           ( COUNTRY=CANADA
               (ACTIVITY OF COMPANY=SITUATION/FINANCIAL/
                  PERIOD OF TIME=800101 A 800930
                    ( (ELEMENT=/
                        PROFIT=123900000
                        MONETARY UNIT=CANADIAN DOLLAR
                      (ELEMENT=/
                        TURNOVER=.10450E 10
                        MONETARY UNIT=CANADIAN DOLLAR
               (CONCENTRATOR=EXPLOITATION
                  PERIOD OF TIME=800101 A 800930
```

```
(MATERIAL=CONCENTRATE
      (ELEMENT=ZN
       PRODUCTION=437300
       UNIT OF MASS=T
       (........)
```

REFERENCES

BAUZOU C. "Interrogation d'une base de données auto-structurante SIGMINI"
Thèse de 3ème cycle Faculté d'Orsay (Avril 1983)

BRISBOIS C. "Méthode de structuration des informations et composition des dictionnaires pour les banques d'information technico-économique et technique"
Union Minière, rapport interne.

LENCI M. "Les Bases de la Codification Sémantique"
Ecole des Mines de Paris - DCI.265.03.74

MORDINI P. "SIGMI, un modèle auto-structurant de base de données"
Thèse 3ème Cycle, I.P. Paris VI (12 Décembre 1979)

Address for Correspondence :

C. BRISBOIS
Union Minière
avenue Louise 54, bte 10
B-1050 Brussels, Belgium
tel : (322) 517 12 15

P. MORDINI
Ecole Nationale Supérieure des Mines de Paris
CAI, rue St. Honoré 35
F-77305 Fontainebleau Cedex
France
tel : (331) 64 22 48 21

Office Data Base Services in an UNIX architecture

AMEDEUS PROTOTYPE

Pietro Barda
Antonio Buongiorno
Franco Calvo
Bruno Pepino

(Olivetti IVREA Italy)

Abstract

Classification, filing and retrieval are important but time expensive activities in an office environment.

Attempts to reduce times, and thus costs, involved in these processes are gaining importance for a better information management in an office. The paper outlines a solution for filing and retrieving "office objects" in an architecture based on UNIX servers and networks of personal computers. The focus is on the data model that supports the Office Data Base and allows a very fast retrieval of structured and not structured information.

In the first section the general requirements of an office data base is described together with a possible realization, in the second an overview of the of the prototype package AMEDEUS, as well as some detail on the current status of the implementation is given.

Index

1. Introduction

The main activities of today's offices, whether or not electronic office systems are used, are:

a) Creating and revising documents.

b) Distributing documents. Documents may be distributed to users via internal or external mail, hand delivery, or electronic means.

c) Filing and retrieving documents. Documents may be filed in and retrieved from file cabinets, libraries, or electronic mass storage.

This paper uses the word document to refer to the user created information that flows through and between office systems. The concept of document includes many kinds of information, such as messages, reports, memos, letters, contracts and so on, not ordinarily thought of as documents.

The automation of offices is becoming a reality for an increasing number of organizations. Office automation is helping these organizations to improve the productivity and effectiveness of office workers and to improve the timeliness of the information on which they depend.

By using computer tecnology, office systems offer the potential for many other functionalities, not just faster typing, for example the ability to integrate data files with text, store and retrieve information distribute documents electronically. The above activities are office automation capabilities. While some of the benefits of electronic document processing can be realized from a single, stand alone office system, a network that interconnects several office systems can bring greater gains in productivity. Physically, a network consist of interconnected equipment and software used for moving documents between offices where they may be created and used. For the user, a network is a collection of services to create, revise, distribute, file and retrieve documents.

Office systems offer different services accordingly to the needs of different users; the entities that tie the systems together are the distribution service and the information interchange (data format conversions).

Unix Office Integrated Environment prototype (called UOIE in the remainder of this paper) proposes a systematic approach to carry on these office activities.

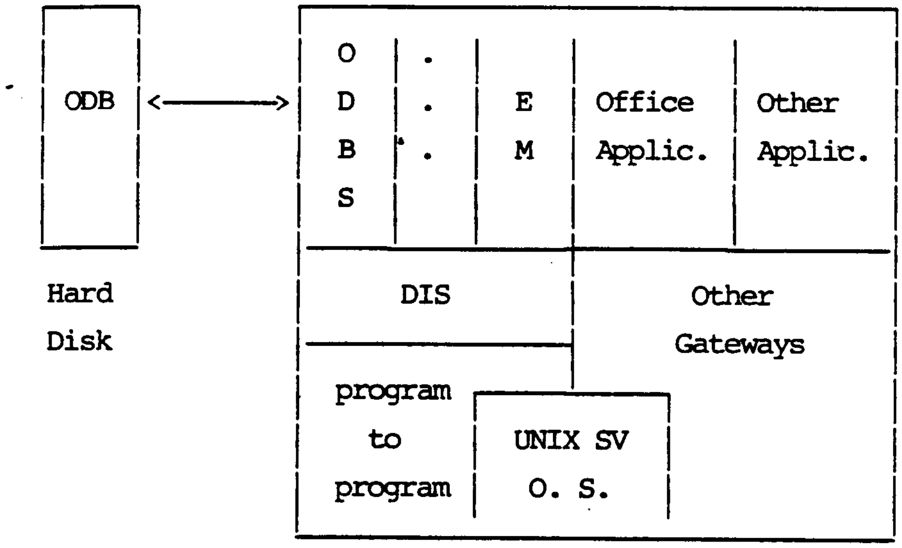
UOIE offers the following departmental cooperation services:

- A centralized information filing and retrieval system (Office Data Base Service - ODBS), the main topic of this paper.

- A network distribution system (Electronic Mail - EM).It is a distributed message system for the exchange of interpersonal mail. Users of EM can invoke it on local computers, prepare mail, and send messages to a user on a different computer, however, gateways allow interchange with mail systems such as UUCP and so on. This network distribution system follows the international CCITT recommendations for Message Handling Services in term of mail format and protocols.

- A document interchange system (Document interchange system - DIS), It involves the interchange of a document to be filed in and retrieved from the ODBS and the interchange of a document to be distributed among users. DIS handles the different naming convenction and takes care of transforming the information form (data stream) used by the supported product.

UOIE - 3b2/400 SERVER



In the remainder of this paper we describe the main concepts and the requirements for office data base services (ODBS).

2. ODBS: requirements and basic concepts.

Before examining the ODBS's concepts, we describe the main requirements for office information retrieval system and workstations.

A general purpose database for an office system based on a server(a minicomputer or a powerful last generation personal computer) and on a set of intelligent workstations has many requirements that are different from those of a typical Data Base Management System (DBMS) designed for data processing on a mainframe computer.

Today's trend is to have more memory on a workstations in order to implement more advanced functions such as sophisticated user system interface. As workstations are used mainly for iteractive applications, good response time is more valuable than high throughput.

In an office environment intelligence (cpu) are generally more distributed than in a classic DP environment but these processing capabilities are mainly devoted to the typical interactive applications that require always more user friendly interfaces.

The main drawback of personal computers is typically a slow I/O; moreover an operating system supporting single tasking does not help to improve the global throughput.

Another important factor is the amount and the location of secondary storage in an office system in comparison with a host.

On the other hand a lot of database activities are I/O bound and require large concentrated storage.

All these considerations imply new criteria in design choices to obtain good global performance optimization.

Office information retrieval systems are developed to help analyze and describe the information stored in a file, to organize them and to retrieve them in response to a user query. Designing and using a retrieval system involves the following main activities: information analysis, information organization and search, query formulation and results presentation.

The main differences between the ODBS and the DBMS are:

1) the data structure,

2) the query language,

3) the operational requirements, update frequency and size of database.

The data structure of a DBMS consist of objective attributes, such as a person's name, age, salary,that could be used to identify a personnel record in a personnel file; similarly in a library file a papers's or book's author, publisher, and date of publication could be used to identify the record of paper or book.

Many access methods have been studied for structured records (objective attributes): tree structure, multiattribute hashing and so on.

On the other hand typical office data (that we call documents and that can be letters, forms etc.) are not structured at all, thus an ODB is not structured and utilizes subjective attributes or content terms, to describe each stored information.

Thus, ODB consists of words or phrases describing the content of the document. At first glance nothing corresponds to objective attributes; so, the only possible access method is the full scanning of the text. Obviously, this method is slow and expensive because ODB can be very large.

The need for a more structured representation of documents has suggested the following method; during document analisys one creates a list of key words excluding a standard common word such as propositions, conjunctions, articles and other non significant words taken from a concordance listing.

This step of text analisys is called "indexing".

The list of key words associated to each document is viewed as the attribute value more closely resembles:a formatted file or relation in terms of relational DBMS , so , despite some differences, is possible to use the same access methods.

Generally, we could divide the query languages for text retrieval into two categories. The first one, influenced by DBMS, is based on Boolean algebra. This type of query language consists of words or parts of words, linked with the Boolean operators such as OR, AND, NOT. This query language is useful when users have a clear idea about what they want.

The second class of the query languages involves the answer relevance (ability of the system to satisfy the user). The answer relevance involves more complex considerations and factors such as system evaluation. There are different components that affect the system evaluation such as:

- User population, type of user, rate of requests, etc.
- Collection, type of documents available at input, coverage of collection.
- Indexing, type of index, level and accuracy of indexes, etc.
- Analysis and search, type of searching, power and complexity of search mechanism and accuracy of search.

The basic factors to be measured to estimate the answer relevance are:

- The recall of the system,that is, the relevant documents retrieved in answer to a search request.

- The precision of the system, that is, the set of documents retrieved that is really relevant.

To obtain these measures is necessary to divide the document collection into four group; retrieved, not retrieved, relevant, not relevant; hence the corresponding definitions are the following:

Recall = (num.documents retrieved and relevant) / tot. relevant

Precision = (num.docs. retrieved and relevant) / tot. retrieved.

In this type of query environment there is no clear distinction between documents that qualify and documents that do not; some documents are really relevant, while others even if present in the result are less or no relevant at all.
In few words when using this class of query languages documents are ranked according to their degree of relevance, and the most relevant are returned to users.

About the operational requirements the important thing to consider is that the office environment is dynamic, and the access method has to be more flexible as far as insertion is concerned. Moreover, the access frequency of documents changes with the age of the document, it generally decreases with time.

From the above considerations it comes that a feasible approach to an Office archiving and retrieval system is one we call "central ODB/distributed desk".

We introduce in this way two entities: a central archive (ODB) and a personal work area called desk. It is worth repeating that our main concern is to share documents between users of probably non-homogeneous work stations and utilize in the most efficient way the global processing capability of the system.

This approach allows:

- the sharing between users of an archive collecting objects of different type but easily interchangeble (by mean of services provided by DIS)
- the sharing between users of an archive hetherogeneus (for we archive different types objects) but with an homogeneous security service
- the clear division between common archive and personal work area mantaining the integration of the two.

The desk can be identified with some environment monitor (topview like) or with some special purpose package running on PC or on the server and using tty. The desk is a personal work area where office objects are created, modified and printed. Every user has his own desk.

All office productivity tools and the functions of archiving and retrieval are activated from desk. We call "local agent" the part of desk that manages the interface between desk and the central ODB.

The concepts of desk and ODB map the following organization of an OA environment:

- each user creates and modifies documents on his desk (electronic or traditional) using the office productivity tools (wp or paper and pencil).
- productivity tools must be called from desk; documents must be in the desk to be manipulated.
- only "current" documents (you are editing or you have just drafted) reside on desk so they can be easily identified simply by name. Sophisticated retrieval systems are not required in the desk.
- it is up to the user free the desk erasing obsolete documents or archiving them.
- the organization of the desk is up to the user and not dictated by a central policy.
- each user has associated a login name and a password.
- documents in the desk are not shared; there are two ways to grant

other user the right to manipulate them: to do an operation of "change desk" in wich all the rules of the authorization system are applied or to archive the document in the central ODB.

- the retrieval of a document consists in a search in the ODB and eventually in a copy of the selected documents in the desk where they can be modified.

Up now we have used the words object and document in an interchangeable way; more precisely we define an object or document as the union of a structured part called profile that describes it and distinguishes it from the other of the collection, and a non structured part called body. All the profiles have the same structure while the bodies can greatly differ, (wp texts, ascii texts, spreadsheets ...).
Each document has associated a system type qualifier (defining documents created by different tools dealing with different data formats) and a user defined type that characterizes a particular usage of the document in the office (letter,memo,balance sheet ...).

The two main services ODBS provides are archiving and retrieval:

- the archiving function involves three steps:

  1. the document profile preparation (on the desk)

  2. the keyword indexing of the document body and insertion of the keywords in the ODB structure (this operation involves a processing activity in the desk and a data transfer toward the ODB)

  3. the physical transfer from desk to ODB of document profile and body (in its original format)

  According to the different types od documents the process of keyword indexing can not be applied to the body (images,spreadsheet...) and the insertion in the ODB is limited to the structured part of it.

- the retrieval service gives to the user the capability of getting from the ODB a set of documents matching a search criteria. Three are the steps needed:

1. query formulation (on the desk); the search criteria concern both the profile and the body. The logical operators AND,OR and NOT are supported in a transparent way to the user, that is they are implied by the structure of the language (presented by form).

2. search processing (on the ODB): a query optimization module drives the search opersations to get the result. The result is a set of document profiles identifiers.

3. retrieve process involves the transfer of document bodies and profiles from ODB to desk.

All these operations are somehow bound to the security mechanism that is not described in this paper.

3. What is Amedeus

Amedeus is a prototype of a full content document retrieval system for the office. It provides functions to analize, organize, store, search and retrieve documents.

Over the last several years, document retrieval systems have received an increasing amount of attention because a semplification of the information handling problems becomes more urgent, but also because the technology appears to provide the necessary means for generating acceptable commercial solutions.

Amedeus is designed for professional offices using documents created and located anywhere in the file system of an UNIX System V.

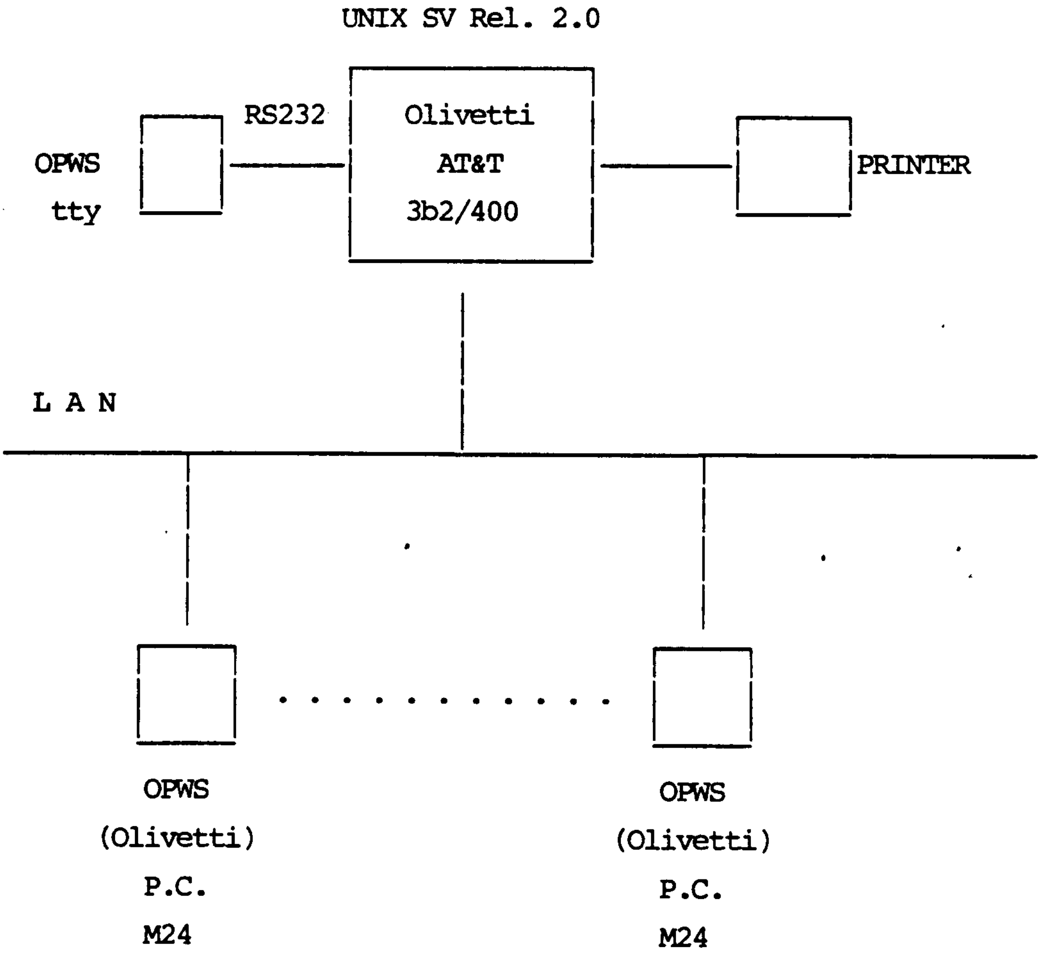
It is a part of a research effort to develop a basis for office integration and eventually to support a distribution service (electronic mail) and office procedure service.

Amedeus is only a reasearch prototype system; no future Olivetti product is implied by this paper.

This system does not provide tools to create or modify documents, in fact it is not strongly integrated with some commercial word processing, rather, the package has been designed to allow the integration with the best sellers, existing word processors running under UNIX System V.

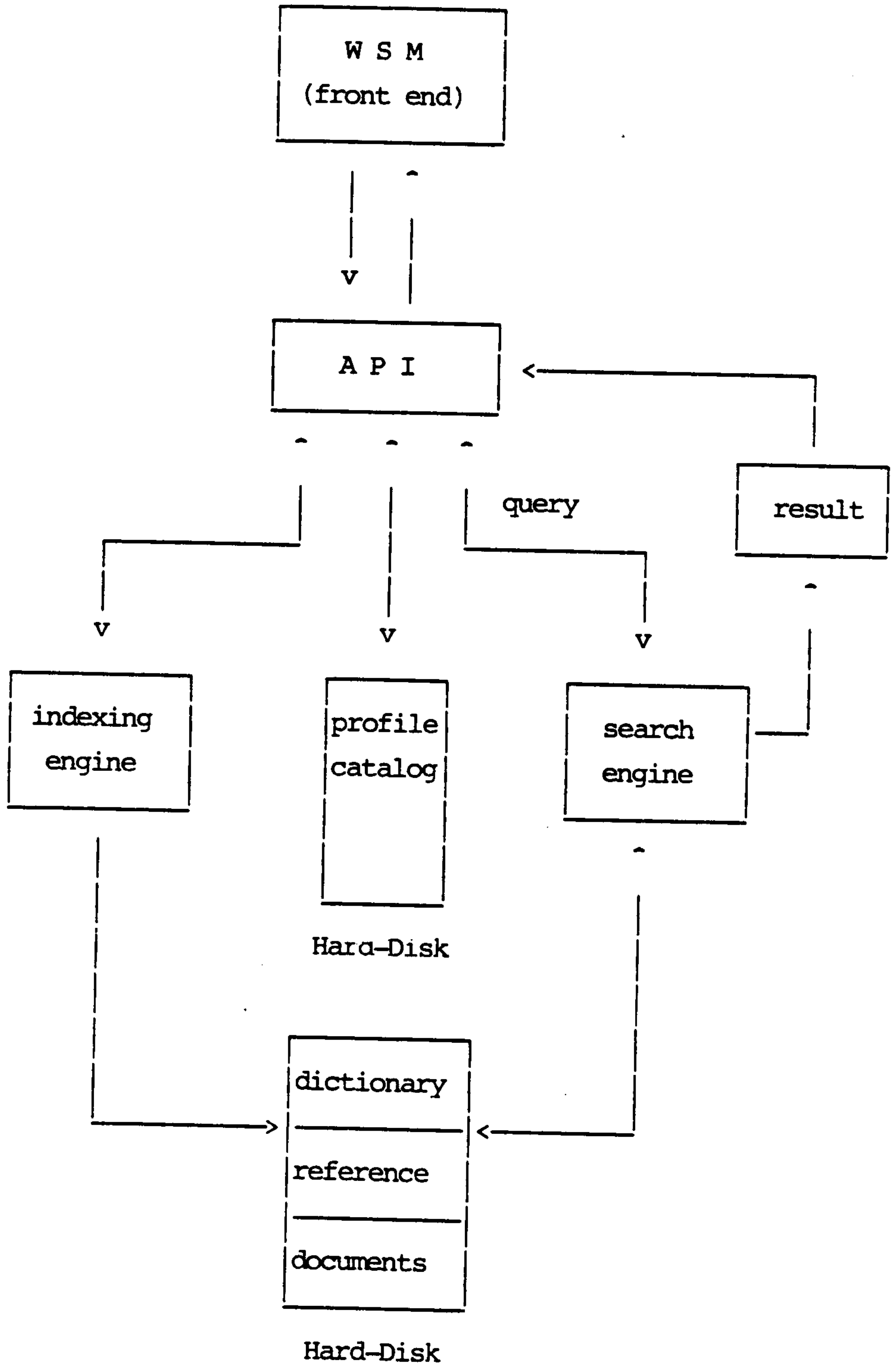
Amedeus is organized as a server of a set of office personal work stations (OPWS) distributed in different offices.

The following figure shows the basic system configuration; a number of semplification have been made for the sake of clarity.



4. System Overview

In this section, we describe the basic architecture of Amedeus, the following schema illustrates the principals components of the system. It is important to point out that for shortness the schema and the related descriptions refer to a general case desk + ODB and all problems related to server-PC interconnection ,included the program to program protocol necessary, are not covered by this paper.



Through the Work Station Monitor (WSM, front end) the user can issue commands interactivly and examine the storage subsystem, an "on line" help facility is provided.

WSM interfaces the API (Application Program Interfaces) that provides the program interface to the "low level" storage subsystem.

Indexing engine is composed by two layers: the text analysis layer and the "write database" primitives layer.
Text analysis module processes the document to extract the keywords and determine their type in order to insert them in the ODB by means of the write primitives adopting a suitable strategy.
Altough full document inversion seems very expensive in terms of storage, an appropriate data model and efficient compression algorithms, can reduce the overhead to an estimated 50% of the original document.

The search engine too is composed by two layers: a query optimization layer and the "read database" primitives layer.
The query optimization module is responsible for the query analysis and for the implementation of an efficient strategy of search depending on the query semantic. The "read database" primitives access the database to fill the result structure to be returned to the WSM.

The ODB data model consists of:

- profile files ; a variable length record file containing the document profile and an associate reference file.

- index files; they are organized as an ISAM (indexed sequential access method) structure with dinamic index. This organization gives all the benefits of a sequential organization and in searching the benefits of random access methods. It is based on a dictionary file where the keys are stored in order and in pages consecutive (physically or logically) and on a index B*-tree lacking the leaf nodes.

5. Getting started with Amedeus.

After you have installed Amedeus, it is easy to begin working with it. The front end component WSM of Amedeus divide the main screen into three parts:

1) The text area takes up most of the sreeen.

2) The command line at the bottom of the screen shows you the various things you can do with Amedeus.

3) The messagge line displays information and errors below the command line.

The documents you want to work with are located in the desk (in this implementation the file system).
To tell Amedeus what to do, you choose one of the commands in the command line. These commands are the following:

Insert: Amedeus reads, selects keywords, and inserts the keywords list in the ODB. We can insert any document made on programs that use ASCII data format and control characters, including Olivetti word processors.

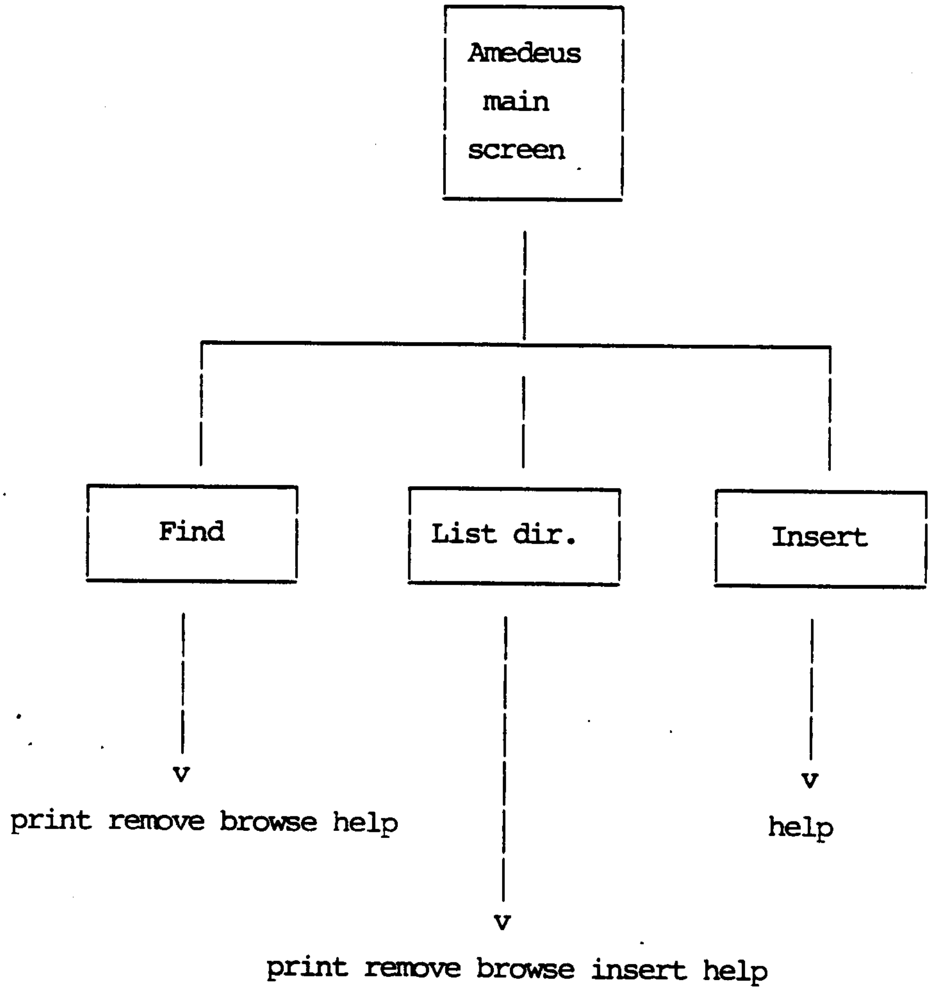
Find: Amedeus searches the search terms into ODB.

List-directory: Amedeus shows you a list of documents stored in the desk (current directory).

Help: Amedeus shows you the Help file.

Quit: Amedeus returns to Unix.

The following figure shows the commands provide by Amedeus.



6. Conclusion

A version of Amedeus is currently running on a set of 255 documents, about 2Mb disk space. This collection was chosen from a real document set; it served to analyze the growing rate of the data model in an highly lexically specialized environment. In this environment we have seen a fast saturation of the dictionary.

Due to a high performance design, Amedeus usually has a response time of less than a second for a typical query.

THE DESIGN OF A SYNTAX-DIRECTED TEXT EDITOR

Brian Collins

Imperial Software Technology

ABSTRACT

The editing and user-interface system described in this paper appears in many different guises to the user. Among these are:

1. A general-purpose, multi-file, multi-windowing screen editor for ASCII text files and simple terminals.
2. A forms system for constrained data entry and presentation.
3. An interactive windowing front-end to application programs.
4. A syntax-directed editor.

This paper concentrates on the latter.

In the design of such an editor, the choice of facilities offered to the user is often more difficult than the actual implementation. In this editor, the decision taken was to make the interface appear to the user as close as possible to a standard text editor, hence the more descriptive title of 'syntax-directed text editor'. The overall effect is to provide a text editor in which it is impossible to enter a syntactically incorrect program.

The editor is language-independent (indeed, it may handle many different languages simultaneously). The syntax of the language is described in an extended, annotated BNF, which also specifies the layout rules for line breaks, spaces and indentation.

The paper provides a description of the editor, both from the user's view and from that of the language implementor. It describes how the syntactic structure is maintained in what is essentially a text editor, and the techniques used for the fast re-parsing of edited text. Finally, the paper assesses the applicability of syntax-directed editing to various languages, its use in a programming environment, and a specific analysis of the problems in the syntax of the language C.

1. INTRODUCTION

A syntax-directed editor can be said to have succeeded in meeting its (supposed) objective of usability if a significant proportion of potential users prefer to use it instead of a conventional text editor. On this basis, many, if not most, of the current syntax-directed editors should be classified as failures. For example, many such editors require expressions to be entered in reverse polish, or require large blocks of text to be frequently reparsed.

The **synde** editor was designed to reduce the risk of failure by providing to the users what they appear to require, which is a text editor in which it is impossible to enter syntactically incorrect programs. This is achieved by treating language syntaxes in two distinct ways: larger multi-line objects are maintained by synthesis, but smaller single-line objects may be freely edited and are maintained by analysis.

Synde was derived from an existing general-purpose screen editor, developed within IST and used as a component of the integrated project support environment, Istar. This editor supports multiple files, windows, terminal types supported by termcap or **terminfo**, and has a rich repertoire of editing commands and operations. A feature of the editing style is that there are no hidden modes (moving, inserting, deleting, command, etc) and therefore all keystrokes have easily predictable results. Printable text typed at the keyboard is always inserted into the document. Arrow keys and function keys are used to move the cursor, to delete, move and copy text and for more complex editing functions.

This editor had already been extended for Istar to support a forms system for constrained and validated data entry and to support menus for option selection. Within a form, the user is allowed to move the cursor freely and to insert or delete text within fields as though the form were a normal text document. However, there are two restrictions. The cursor cannot be moved onto any protected ('preprinted') text on the form. Whenever the cursor is moved off a field, that field may be validated by some external mechanism (unknown to the main part of the editor), which handles the 'event' of moving between lines.

This notion of text editing, with external event handlers to manipulate a structure unknown to the editor is fundamental to the design of **synde**.

2. REQUIREMENTS

The requirements for **synde** were mainly self-imposed and included:

- That it support any 'reasonable' programming language. 'Reasonable' was taken to mean that the syntax of the language could be described with an extended Backus-Naur Form (BNF) and the lexical structure could be described by regular expressions. These are similar to the the constraints imposed on languages supported by **lex** and **yacc**.
- As far as possible, it should 'feel' to the user as though it were the normal screen editor acting on a text file. Keys which invoke editing operations in the text editor should act in the same way or analogously.
- It should support multiple files and windows and allow the user to work with more than one language at a time.
- Support for specific languages should not be built in, requiring recompilation to add a new language. Instead, the necessary information describing the language should be dynamically loaded on demand.
- The languages should be describeu by a notation that is easy to read and write.
- The performance, when editing any syntactic structure, should be similar to editing with a text editor. There should be no significant delays for any operations.

Equally important are the areas that **synde** does not attempt to address:

- Semantic analysis.
  The semantic analysis of a program in an arbitrary language, as the program is being edited, is an unsolved research problem. No attempt is made to handle any semantics.
- Fast compilation.
  As **synde** is designed to be language independent, it can have no knowledge of the parse-tree structures expected by a compiler back-end. Therefore it can only act as an editor; a textual representation of the program must be passed to the chosen compiler.

3. SYNTAX DEFINITION

A language supported by **synde** must contain some structure that is both used and enforced when editing. This structure is handled as a number of layers.

3.1 Lexical Structure

A *document* (usually a program) in the language is composed of zero or more lines. The lexical structure of each line must be described by a context-free grammar, represented as a set of *regular expressions*. Using this grammar an isolated line can be unambiguously split into zero or more *lexemes*, which are the terminal symbols of the syntax. No lexeme may span a line boundary. A line break is always significant.

3.2 Comment Structure

Some characters (eg spaces and tabs) may be specified as being ignored wherever they occur, except where explicitly allowed within some larger lexeme (eg in a string). These are not treated as lexemes in their own right, but just as *whitespace*.

Similarly *comments* are lexically distinguished lines or parts of lines that are not formally part of the language and are to be ignored wherever they occur. **Synde** supports comments that fall into one of two classes:

— *Horizontal comments* appear at the end of a line that may contain other text.

— *Vertical comments* span one or more complete lines, which may contain no other text.

Almost any programming language uses one or both of these classes of comment.

3.3 Syntactic Structure

The syntax for a language supported by **synde** is split into two classes:

— *Horizontal Syntax* is that part of the language that does not, inherently, span multiple lines. This does not preclude a *horizontal construct* from spanning more than one 'continuation lines' in individual cases. Typical horizontal constructs are assignment or procedure calls.

— *Vertical Syntax* is that part of the language that necessarily spans multiple lines. A *vertical construct* 'contains' other horizontal or vertical constructs. Typical vertical constructs are 'while' statements or procedure definitions.

A *slice* is either a complete horizontal construct or a part of a vertical construct that is not part of any other enclosed construct. It is a horizontal 'slice' through a construct, normally consisting of a single line but with possible 'continuation' lines. A typical slice would be the condition line of a 'while' statement or a procedure header. It can be seen that a document is a sequence of slices, each of which contains a small number of lines (usually one line).

3.4 Syntax Definition Notation

The various structures described above are not provided explicitly by the language implementor. Instead, they are derived from a extended, annotated Backus-Naur Form (BNF) definition of the language.

The BNF is extended in two ways:

— By notation which more conveniently specifies conditional and repeated constructs.

— By notation to specify the comment and lexical structures that form the terminal symbols of the language.

The annotation serves to specify the layout and indentation rules for the language. These rules are embedded in the syntax and form part of the language definition.

Figure 1 shows an example of a trivial language definition.

Figure 1. Trivial Language Definition

```
/* Comments */
#                   =     ' ' ;
#                   =     '/*' '*/' ;

/* Lexemes */
identifier          =     "[a-zA-Z][a-zA-Z0-9_]*" ;
number              =     "[0-9]+" ;

/* Syntax */
program             =     statements ;

statements          =     ( statement / )+ ;

statement           =     while_statement
                        | if_statement
                        | assignment ;

while_statement     =     'while' # expression # '{' /+4
                              statements /-4
                          '}' ;

if_statement        =     'if' # expression # '{' /+4
                              statements /-4
                          ( '}' # 'else' # '{' /+4
                              statements /-4 )?
                          '}' ;

assignment          =     identifier # '=' # expression ;

expression          =     identifier | number ;
```

4. USER VIEW

Synde is derived from a text screen editor. In common with many other 'what-you-see-is-what-you-get' (wysiwyg) screen editors, text typed on the keyboard is inserted into the document being edited. Function keys or special key sequences are used for other editing operations. These operations include cursor movement (in units of character, word, line, page or file), text deletion (in units of character, word or line), text cut and paste. In addition, there is a command line on which a richer repertoire of commands may be entered for file manipulation, searching and substitution, parameter setting, etc.

In **synde**, most of these operations have exactly the same effects as in the corresponding text editor. Other operations have analogous effects that are easily understood by the user. A small set of additional operations are only applicable to syntactic editing and are meaningless for text editing.

4.1 Unchanged Operations

The operations of text insertion, cursor movement and text deletion within a line appear to the user as in the text editor. The text on any line may be freely edited. Similarly, commands entered on the command line have the same effect as when text editing.

The only significant difference with operations of this class is the side-effect of syntax checking: correction and completion when the cursor is moved between lines.

Whenever the user attempts to move the cursor off a slice (usually a single line) that has been modified by the insertion or deletion of text, the slice is checked to see if it still conforms to the appropriate syntax. This check takes an insignificant length of time. If the check is successful the cursor moves and no effect is visible to the user.

If, however, the check is unsuccessful, the slice is syntactically incorrect within its context. In this case the editor attempts to correct the error, either by the addition or deletion of text.

The text added is either strings of the language (punctuation or keywords) or syntactic place-holders called *stubs*. Stubs are distinguished on the screen by underlining, and are the names of the non-terminals used in the syntax definition. Subsequent editing of a stub causes it to be automatically deleted. In any syntax check of a slice containing stubs, the stub itself is treated as perfectly valid in its context. Consequently stubs need not be filled in immediately.

The text inserted by the editor has the layout specified in the syntax definition. This may span more than one line with the correct indentation being maintained. For example, in Pascal at a position where a statement is expected, if the keyword 'WHILE' is typed and the cursor moved off the line, the one line of the statement would be replaced by the three lines:

```
WHILE condition DO BEGIN
      statement
END
```

The condition and statement stubs may be completed at any time.

If the slice cannot be completed solely by the addition of text, trailing text will be deleted and new text added to produce a syntactically correct slice. However, only stubs and fixed strings of the language will be automatically deleted as these are easily regenerated when necessary. Significant text, such as numbers or identifiers, will not be deleted. In the case where the slice cannot be corrected without the deletion of significant text, the cursor will be positioned at the best estimate of the place of the syntax error and the error reported to the user. The error must be corrected manually before the cursor is allowed to move from the slice.

4.2 Analogous Operations

Editing operations which may manipulate more than one line in the text editor are inapplicable in editing syntactic documents. More than one line may be rendered syntactically incorrect by such operations on a syntactic document. Consequently, simply correcting the single affected line or requiring the user to correct the errors before moving the cursor will not work. However, there are actions which are syntax-preserving which have an 'analogous' effect to the operation in the text editor. The intention is that such operations perform useful actions, but are unsurprising to a user knowing the text editor.

The RETURN operation in the text editor (normally invoked by pressing the 'Return' key) splits the current line at the cursor position, inserting a new line. Depending on the cursor position, this new line may be a blank line before or after the current line or the text on the original line may be split between the two lines.

In **synde**, RETURN adds a new syntactic construct of the appropriate type for the context in which the RETURN is operated. Referring to the previous Pascal example, operating RETURN on the line containing statement at the beginning or end of the line would add a new statement before or after the line respectively. A separator will be added if needed (in the Pascal case, a semicolon) and the correct indentation will be maintained. As well as inserting a new instance of an existing construct, RETURN will also add an omitted optional construct if appropriate.

Operations in the text editor which DELETE, PICK ('cut') or PUT ('paste') complete lines are redefined to handle complete syntactic constructs. For example, PICK applied to the Pascal 'WHILE' statement above would move the three lines from 'WHILE' to 'END' into the buffer, instead of just a single line. Any other statements that had been added between these lines would also be moved.

Extra constraints are imposed on the use of the buffer for moving and copying text to ensure that a syntactically incorrect document cannot be created. The buffer holds its constructs without indentation. They acquire the prevailing indentation of the context into which the buffer content is PUT ('pasted').

4.3 Syntactic Operations

Synde has a small number of operations that are only applicable to syntax editing. There are operations to navigate the document using the language structure, moving the cursor to the enclosing, enclosed, preceding or following construct. There are also operations to move to the following or preceding stub. Finally there is an operation to *fold* an arbitrarily large construct from the screen, replacing it with a one-line stub. Such folds may be nested and can reduce the amount of unwanted detail visible on the screen. This FOLD operation, acting on an already folded construct, will unfold the stub, revealing the previously hidden text.

5. MAINTENANCE OF SYNTACTIC STRUCTURE

5.1 Syntactic Files

The files manipulated by **synde** are not stored as simple text, but as *syntactic files*. These syntactic files consist of a mixture of text lines and control lines. The text lines contain the whole text of the document. This text is stored in no other place.

The control lines are of one of three types:

— The initial line, which specifies that the document is syntactic and names the syntax definition file for the language.

— Grouping lines, which bracket folded constructs in the document. These lines, saved with the file, allow a folded document to be written out and subsequently re-edited with the folded constructs remaining off the screen.

— Structure lines, which contain the structuring information for the preceding text line or lines.

These lines all start with a non-printing control character but are only treated specially if the first line of the document is a control line. All other lines that do not start with this control character are text lines and contain normal ASCII characters. Highlighted stubs are stored in the document as characters with the 8th bit set. This restricts implementations of **synde** to machines with a 7-bit ASCII character set.

5.2 Structure Records

Languages supported by **synde** are described by a syntax definition. This definition generates a compacted representation of the syntax stored as a directed graph. In this graph, each node can be classified into:

— A line break.

— A 'horizontal construct' node that contains no line breaks directly or indirectly (eg an expression).

— A 'vertical construct' node that directly or indirectly contains one or more line breaks.

Vertical constructs can in turn be broken into:

— A concatenation of two or more sub-nodes.

— A choice (the '|' operator) between two or more sub-nodes.

— The repetition or option (the '*' or '+', or '?' operators) of a sub-node, possibly with a separator (eg a semicolon).

In the compacted syntax representation, every vertical node is identified by a unique number. For a node which is a concatenation of sub-nodes, each component sub-node is either itself vertical, in which case it has its own number, or it is horizontal, in which case the component can be identified by the enclosing node's number and an index into the concatenation. Such a node, which is a horizontal component of a vertical node is known as a *phrase*. Every phrase has a unique two-part number (parent node number + index).

Each slice in the document is followed by zero or more structure records, each specifying information about vertical constructs on that slice. For each such construct, the structure record contains the node number of the vertical construct, the index into the parent node (if appropriate), the depth of nesting of the construct within the document, and the indentation associated with the construct.

The structure record may either refer to a phrase, or to a vertical node. The latter case (distinguished by negative index values) can be of a variety of types:

— Complete vertical construct.

— Folded vertical construct.

— Start of alternative construct.

— Position of omitted optional construct.

— Start of optional construct.

— Start of instance of a repeatable construct.

— Start of separators between repeatable constructs.

— End of instance of a repeatable construct.

5.3 Lexical Analysis

Lines are lexically analysed in isolation, without regard to the syntax. A finite state machine, automatically generated from the syntax definition, splits the line into lexemes, passing from left to right. The algorithm always identifies the longest string possible for any lexeme. The lexemes are classified into:

— Whitespace (to be ignored)

— Comments (to be ignored, but preserved)

— Fixed strings (such as punctuation)

— Regular expression strings, matching a regular expression.

— Erroneous strings.

The special case where the same lexeme matches both a fixed string and a regular expression is known as a *keyword*. It usually applies where keywords of the language are reserved identifiers.

Erroneous strings are created as lexemes in their own right, causing the parser to report them as syntax errors.

5.4 Parsing

When a slice is checked after an edit, it is first lexically analysed and then it is reparsed. The parser takes each structure record for the slice in turn and parses the line as the concatenation of its phrases. Obviously, this concatenation need not be a well-formed construct of the syntax, but each individual phrase will be. The parser uses a recursive descent algorithm directly on the syntax as specified by the language implementor. The parser will look ahead an arbitrary distance to resolve ambiguities, but the look-ahead is bounded by the end of the slice.

The main advantage of using recursive descent rather than an LALR method as used by **yacc** is that the non-terminals are all named by the language implementor (presumably mnemonically), and no unnecessary nodes need be added. This is very important in automatically correcting errors, allowing any syntactic stubs to have suitable names.

5.5 Error Correction and Reporting

When the parser discovers that a slice does not conform to the syntax of its constituent phrases, it attempts to correct the line in a number of ways:

1. If the slice can be completed by the addition of text, this is added on-the-fly by the parser. This text will consist of punctuation, keywords, whitespace and stubs. The smallest amount of text necessary to complete the phrases is added.
2. If a previously omitted optional construct could validly match the text of the slice, that optional construct is added.
3. If all text after the position of the syntax error consists of punctuation, keywords, whitespace, stubs and comments, that text is deleted, and the line re-parsed to be automatically completed as in (1) above. A deleted comment is remembered and replaced at the end of the (now correct) slice.

If all the above fail to correct the error, the cursor is positioned at the high-water-mark of the parse attempt, and the error is reported to the user.

5.6 Vertical Structure Manipulation

The operations which insert, delete, copy, move or fold complete constructs also use the structure records associated with slices. In particular, the structure records which give the positions of the start and end of instances of repeatable and optional constructs delimit lines to be deleted, copied, etc. The phrase records, describing the actual phrases on the lines are not used as these operations all operate on one or more complete lines and not on the contents of the lines.

One, possibly contentious, decision concerns the treatment of multi-line block comments. Presumably, these comments should be deleted, moved or copied with the constructs to which they refer. However, a comment could either precede or follow the appropriate construct, and it is impossible to determine the correct action from the context. In **synde**, the assumption is that a block comment *precedes* the construct to which it refers, and when a construct is deleted or copied, any preceding comment is deleted or copied with it. On the other hand, blank lines are presumed to follow the construct to which they belong.

6. APPLICATION TO PROGRAMMING LANGUAGES

To gain any benefits from a syntax-directed editor, the user must be prepared to accept a discipline in programming that is not imposed when using a text editor. There may often be editing transformations that seem simple when described textually, but imply very significant structural changes. The hope is that the benefits gained outweigh the flexibility lost. The user no longer has the freedom to make mistakes.

6.1 Language Restrictions

Synde is language-independent and can support a large variety of language styles. It does impose some restrictions on the languages that can be supported:

— That the lexical structure can be described by a set of regular expressions, and is not dependent on the syntax.

— That comments either appear at the end of a line or on complete lines or blocks of lines.

— That a line break may not appear within a lexeme.

— That the syntax does not depend on any semantic understanding of the program.

— That the syntax definition is not left-recursive, forbidding constructs defined as starting with instances of themselves. This requires the use of iteration to describe repeated constructs rather than recursion. This also has the advantage of appearing more intuitive.

— That all constructs in the language have an acceptable standard layout, suitable for automatic pretty-printing.

— That repeated vertical constructs end with a line break.

— That alternative vertical constructs can be distinguished by the first line only; ie they must differ before the first line break.

When implementing a language that violates some of these restrictions, two approaches can be used:

— The language can be restricted by the imposition of 'house style rules' to produces a subset language that meets the restrictions.

— The language can be extended to a more general language that omits syntactic restrictions that cannot be checked by the editor.

6.2 C

In a Unix environment, the C language is of obvious interest. Unfortunately C has two major difficulties for syntax editing. Both are because the current version of C has evolved, rather than been designed as a whole.

Firstly, C is not one language but two. A C program is pre-processed by the **cpp** program before passing to the C compiler proper. **Cpp** has its own language for symbol definition and conditional compilation. These pre-processor directives are lines starting with a '#' character and are not seen by the compiler. In **synde** the only possible treatment of these two intersecting languages is to treat lines starting with a '#' character as comments. This means that all defined cpp symbols must be well-formed expressions. Conditional compilation is only possible if the juxtaposition of all the conditional alternatives forms a valid C program.

The second problem concerns the use of 'typedef'. The C compiler actually places identifiers defined as types into the symbol table as keywords, modifying the syntax of the language. In the example:

```
#include "foo.h"
foo(x)
{
        f( x);
        return x;
}
```

the meaning may not be obvious if 'foo.h' contains:

```
typedef int f;
```

The solution is to impose style rules requiring type names to be lexically distinguished from functions and variables.

One final reason for not syntax-editing C programs is the large range of text manipulation programs in Unix that can be used to process conventional C source. These would all be lost to users of a syntax editor.

6.3 Some Other Languages

Some common programming languages for which **synde** may be considered include:

Pascal — Pascal implementations have failed to standardise on details of the lexical definition. Almost every compiler has a different treatment of upper and lower case letters, extra keywords and comments. Therefore the syntax definition needs to be tailored to the target compiler. To avoid the problem of a 'dangling else', the syntax always adds BEGIN & END keywords to all constructs. This is no problem for the user as they need never be manually typed.

Modula2 — Modula2 appears to have no problems at all, since the 'rough edges' of Pascal have been removed. **Synde** can support the language in full.

Ada — Again Ada poses no problem and is supported in full.

Chill — In Chill, the layout strategy for some constructs (such as exception handlers) is not immediately obvious. Chill has been implemented with a layout that appears less ugly than some others considered.

BCPL — BCPL has a 'VALOF block' which can allow a (potentially large) block to appear within any expression. This construct cannot easily be formatted automatically. The solution is to restrict the use of VALOF blocks to assignment statements and function definitions.

Algol68 — As with BCPL, any expression in Algol68 may be a full block, and a similar restriction is required.

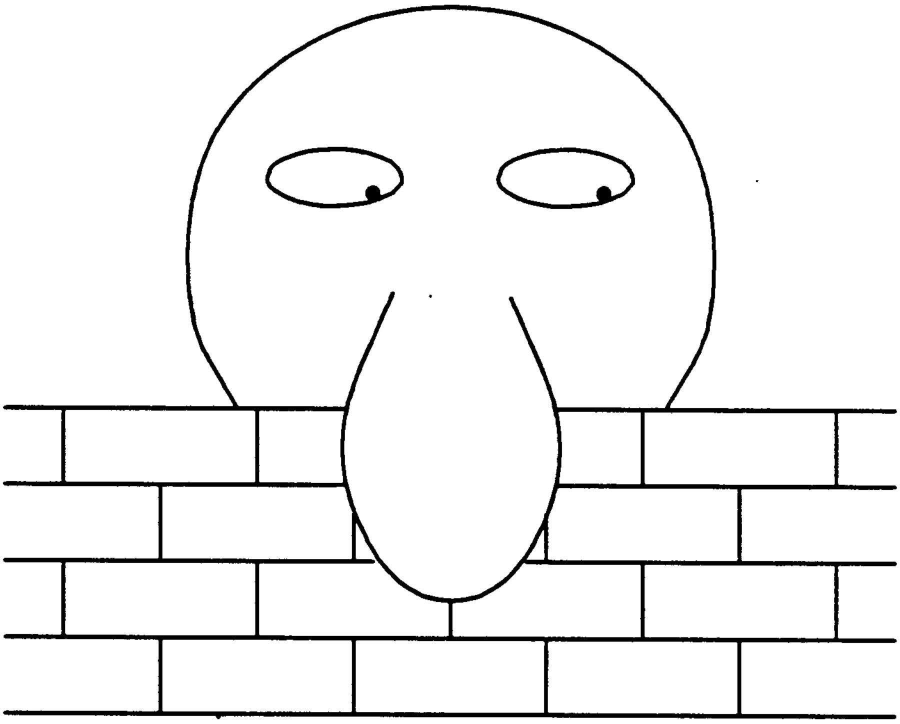
Fortran — Fortran has a lexical structure allowing spaces to appear anywhere. This cannot be supported, but many implementations of Fortran adopt a more suitable format.

7. SUMMARY

The difficulties outlined above in relation to the C language reflect peculiarities of that language rather than inadequacies of **synde**. The editor has now been in use for some months, and has been applied both to specification languages - Meta-IV, Z and SDL-PR - and to programming languages - Pascal, Ada and C. Experience with these implementations suggests that **synde** is convenient for language implementors and attractive to end users. The syntax definition required for **synde** to support a new language can readily be prepared; Ada, for example, was implemented by one person in less than a day. For the end user, **synde**'s distinction between horizontal and vertical syntax offers the major benefits of structured editing without the irritations that can arise when the syntax directed approach is taken to extremes. Because of its convenience and benefits IST expect to exploit **synde** both in the Istar context and as a stand-alone editor.

A Guided Tour of OSI based NFS

Pete Delaney



"Wot, No paper?"

System Management for a Distributed UNIX Environment

Dipl.-Inf. Winfried Dulz
Dipl.-Inf. Roland Langer

Chair for Computer Architecture and Performance Evaluation
(Prof. Dr. U. Herzog)

University of Erlangen-Nuernberg

ABSTRACT

In analogy to system management for central server configurations, distributed systems must also provide utilities for user handling, system accounting and resource accessing.

We show for the UNIX network operating system Newcastle Connection how transaction-oriented protocols can be used to build a distributed tool based on cooperating client/server processes, that will handle the distributed user management without loosing the main characteristic of NC sites, i. e. the local autonomy of loosely coupled UNIX systems.

1. Introduction

The great expansion of our campus-wide UNIX environment leads to many new problems, especially the user management of those students who are using the systems only for programming courses could not be neglected.
Each user of the distributed UNIX-United system, based on the network operating system Newcastle Connection (NC), can be categorized into one of three classes:

1) only local access to one or more nodes

2) local access to some node and remote access to some other node without having local access on that node

3) local and remote access on several nodes

The NC philosophy is to let each local UNIX system have its own system management, i.e the systems work independent of each other and remote access is only possible via a Remote Procedure Call (RPC) protocol if the remote system manager has granted access to his site via the NC mapping function.
It is clear that the management overhead for users of the third class is enormeous, because each user who wants to access N nodes must visit N system managers, each of them has to make N-1 mapping entries, this means N*N entry operations in the worst case.

The goal of our approach was to build a distributed tool based on cooperating client/server processes, that will handle the distributed user management without loosing the main characteristic of NC sites, namely the local autonomy of loosely coupled UNIX systems

This paper will discuss our design concepts, especially the performance and reliability issues and describe some implementation details with respect to UNIX System V. First of all we will give an overview of our UNIX-United system and the major principles of the NC.

2. The Erlangen REVUE Network

3. Design Issues

- cooperating client/server processes
- performance issues
- reliability issues

4. Implementation Details

- process structure
- interprocess communication (IPC)
- system manager interface

5. Concluding Remarks

An Implementation of NFS under System V.2

Bill Fraser-Campbell

The Instruction Set Ltd
152-156 Kentish Town Road, London NW1 9QB

Mordecai B. Rosen

Lachman Associates Inc
645 Blackhawk Drive, Westmont, Illinois 60559

ABSTRACT

Sun's Network File System (NFS) has been ported to UNIX System V. The implementation raises some issues of general relevance to UNIX based network file systems.

1. Introduction

Sun Microsystems' Network File System (NFS) has been ported to UNIX[1] System V Release 2.

The project originally developed out of discussions between Lachman Associates and Sun, and was carried out as a joint effort by our two companies. The system is available in source code for use as a reference release of NFS in System V, and is a fully supported product.

2. Technical Approach

The code is derived from the AT&T System V Release 2 Version 2 source for VAX 11/750. Where possible, changes have been made in such a way as to allow direct comparison of the NFS source with the original. Particular attention has been paid to retaining SVID[1] compatibility.

3. Overview

In vanilla System V, the data structures which represent objects in the file system, "inodes", are manipulated directly by code in the upper levels of the kernel. The code is highly dependent on the design of the System V file store. In the System V NFS kernel, inodes and operations on them have been pushed down to a deeper level, and the upper layers of the kernel deal with file system independent objects called "vnodes"[2]. The code which handles vnodes is called the VFS. Operations in the VFS are not aware of the type of file system underlying a particular vnode.

[1] UNIX is a trade mark of AT&T in the USA and other countries.

user programs	
system call processing	
VFS	
UFS	NFS
drivers	RPC/XDR
discs	network

Figure 1. Kernel layers

The VFS is embedded in the kernel as an interface layer between high level routines and the file system specific code. Changes to the kernel are largely confined to the code around the VFS layer, so that compatibility is maintained with System V system calls and disc formats.

The VFS layer allows the kernel transparent access to any number of different file systems. The System V NFS kernel supports only two at the moment.

1. The UFS, or UNIX File System, the local filestore.
2. The NFS, or Network File System.

The NFS sits on top of Sun's published Remote Procedure Call (RPC) and External Data Representation (XDR) packages. By defining network protocols for remote file operations, the System V/VFS/NFS combination allows System V to use files on other NFS systems as though those files were held on local discs. In addition, System V NFS can act as a file server to other NFS systems.

The principal extensions are new system calls for mounting and unmounting remote file systems. Otherwise, System V behaviour is largely unchanged. Most programs do not even need to be recompiled to gain access to the full functionality of NFS.

For details of the general properties of NFS, we refer the reader to [3].

4. What Was Involved

4.1 The Vnode Kernel

The System V kernel was split into two parts. One contained the System V system call interface, and those parts of UNIX which are clients, not components, of the file system: in other words, all those parts which can be considered as file system independent. The other contained all the code which implements the System V file store, destined to become the UFS.

Any functionality above the VFS can be applied transparently to any of the file system types below the VFS. Operations carried out below the VFS only apply to a single file system. It follows that the decision on where to split the kernel has a major effect on the behaviour of the finished article.

For example, in the System V NFS port, the paging code is positioned above the VFS, making it possible not only to execute remotely stored programs, but also to demand page them over the net.

In contrast, the specification of the *ustat* system call makes it impossible to ustat a remote file system, unless the arguments are changed. Rather than do that, we chose to put *ustat* below the VFS, so that it will still work, but only on local file systems. To allow users to gather information about remote file systems, we wrote *statfs* and *fstatfs* system calls into the NFS kernel. These system calls are defined to have the same arguments and effects as the Sun system calls of the same name. Since they lie above the VFS, programs which use them will operate equally well on local and remote file systems.

Naturally, much of the code above the VFS had to be reworked to use the VFS interface. Most of the changes were of a mechanical nature, such as replacing calls of "namei" with calls of "lookupname".

The major additions to the system above the VFS are:

1. Each component of a path name has to be interpreted individually, since any component might be a remote mount point. The new lookup code is less efficient than namei, but is more suited to the NFS environment, and easier to maintain.
2. To compensate for any loss of performance due to the replacement of namei, Sun's directory name lookup cache has been ported to System V. This cache retains the vnodes which describe recently used directories, so that the number of lookup operations is kept to a minimum.
3. A kernel heap, which is used for all transient storage in the NFS code, has been fitted. Heap storage is freed into a fast-fit tree[4], rather than released, so that subsequent requests for heap store will be processed quickly.

4.2 The UFS

The UFS, or local file system, is media compatible with System V, and much of the code is unaffected by the NFS port. However, NFS requires a generation number to be kept for each file on disc, so that the NFS server can detect accesses to files which no longer exist. System V inodes contain 13 disc addresses packed into a 40 byte space, so we are able to use the 40th byte to hold the generation number. Since one byte can only hold 256 possible values, there is a chance that the generation number could cycle round fast enough for mistakes to happen. In practice this turns out not to be a problem.

Some revisions and extensions were made to the buffer cache. NFS caches data from remote systems in the same buffer pool as local data. This introduces some problems. In particular, in an NFS kernel it is meaningless to associate a buffer holding remote data with either a major/minor device number or a disc block number. Instead, the data is identified by the address of a vnode and a logical block number within the file that vnode represents. Logical block numbers start at zero for each vnode. Using logical block numbers in the cache destroys the usefulness of the hashing algorithm used to locate buffered data. We found an adequate, fast hash function to be the address of the vnode plus the logical block number. NFS makes it necessary to flush the buffer cache more frequently than before. The flushing acts as a slight penalty on the use of remote files.

The most significant change in the UFS code is in the handling of directories. All directory manipulation is now concentrated in one module. The code reflects the new strategy for path name parsing implemented in the VFS layer, and goes to great lengths to avoid deadlocks and process interactions without violating the stateless nature of an NFS server.

The device driver interface is source code compatible with System V. It proved to be impossible to retain binary compatibility as kernel data structures were modified. A representative range of drivers has been built and run without trouble.

4.3 The Network File System

The Sun Network File System (NFS) has been implemented under the System V VFS as simply another file system type. It provides the capability of manipulating files on a remotely mounted file system, transparently. That is, local files and remote files can be operated on with a single set of file system semantics. Both the client and server side of the NFS protocols have been implemented and reside within the System V kernel.

Along with being transparent, the NFS is designed to offer file sharing in a heterogeneous environment. In order to facilitate this, it is forced to view some file system objects like directories and directory entries in a somewhat general fashion. This necessary network view causes some problems for System V.

For one thing, directory entries have to be converted between System V format and NFS network standard format. This causes additional overhead to be incurred but does not introduce any loss of functionality. Though read() and seek() are still supported for directories, there is some semantic sacrifice when they operate over the net. It is clear that seeking to a particular offset within a directory does not have the same meaning that it did in the local case. In addition, the size of a remote directory may not directly reflect the number of entries contained within. In order to provide a more general interface to directories, a new system call called getdirentries() has been provided in System V NFS. Isolating applications from direct access to directories appears to be the direction, being taken for System V.

By design, NFS puts the burden of state maintenance on the client side and allows the servers to be stateless. In the event of a server crashing, a client need only retry an operation until successful. Since clients often want to know the status of a file's attributes, such as size, mode, or modification times, this information is included in the packet transferred for most operations. To counteract any performance penalty associated with the need to have up to date attributes available on the client, an attribute cache is part of the NFS design, and was ported to System V NFS. Though this cache is a definite performance improvement, it can give a somewhat false view of a remote file. It is possible for a stat() system call to return the attributes of a file that is still cached but was removed a few seconds ago by another machine. These potential synchronization problems introduced by the cache are not a problem in practice.

4.4 RPC and XDR

RPC, or Remote Procedure Call, is the mechanism by which the NFS communicates with other machines. This facility provides the NFS with the capability of executing file system primitive operations remotely. RPC and the supporting functions that implement the eXternal Data Representation(XDR) protocols have been added to the System V kernel.

RPC insulates the NFS from any knowledge about the underlying transport mechanisms. The only view that the upper layer has is that of a natural programming interface. Remote calls are made over the net by simply calling a function as if it were to be run on the local machine and within the address space of the invoking process. The fact that the arguments to the function are being converted into a network standard format and the specified function executed on another machine is transparent to the caller.

The System V RPC implementation uses UDP/IP as the underlying transport mechanism. Even though RPC includes a generalized facility for interfacing to more than one transport, in practice UDP/IP is the only one currently used.

The UDP/IP transport layer used in this implementation was provided by integrating Excelan's EXOS 204 package into System V. This package is comprised of an intelligent ethernet controller card running UDP/TCP/IP on board along with an associated driver to provide 4.1C BSD socket primitives. The vanilla package provided us with 90% of our transport requirements. The fact that the only interface to the EXOS socket functionality was through a driver turned out to be somewhat awkward. In the end, a new kernel level interface to EXOS sockets had to be devised.

5. Details of System V NFS

5.1 SVID Compatibility

Departures from the SVID have been kept to a minimum. The following system calls are affected:

system call	status	details
flock	restricted	will not flock remote directories
link	restricted	cannot link to directories
mknod	restricted	cannot make directories
mount	restricted	local file systems only
read	restricted	some restrictions on remote directories
seek	restricted	some restrictions on remote directories
umount	replaced	different argument
unlink	restricted	cannot unlink directories
ustat	restricted	local file systems only

TABLE 1. System call interface changes

The restrictions on *link*, *mknod*, and *unlink* are necessary as remote operations have to be atomic, to reduce the chance of failure in mid-flight. The new system calls *mkdir*, *rmdir*, and *rename* provide clean, atomic transactions which replace the sequences of system calls previously necessary.

In addition, several of the user commands are affected by the NFS changes. All but one of these are a direct consequence of the changed system call interface or changes to the kernel data structures. The exception is "find", which used *stat* to find the length of a directory and deduced from that the number of System V directory entries which could fit into that space. Such a calculation is no longer valid when dealing with directories not in the System V format.

command	status	function
crash	revised	handles changed data structures
df	revised	uses new statfs system call
find	revised	does not stat remote directories
killall	recompiled	depends on kernel data structures
mkdir	replaced	uses mkdir system call
mount	replaced	uses new mount, nfs_mount system calls
mv (cp, ln)	revised	uses rename system call
mvdir	revised	uses rename command
ps	recompiled	depends on kernel data structures
rmdir	replaced	uses rmdir system call
umount	replaced	uses changed umount system call

TABLE 2. User command changes

System V device drivers are compatible with the NFS kernel at the source code level.

5.2 Extensions

Almost all the extensions described are necessary to make NFS work. Some are performance options, and could be omitted if desired, but we found them to be so effective as to be essential. The following system calls are new:

system call	details
async_daemon	asynchronous i/o daemon entry
fstatfs	local and remote file systems
mkdir	necessary to make an atomic operation
nfs_getfh	mount server entry
nfs_mount	remote mount
nfs_svc	NFS server entry
getdirentries	returns a network wide directory format
rename	necessary to make an atomic operation
rmdir	necessary to make an atomic operation
statfs	local and remote file systems

TABLE 3. New system calls

Some new commands are provided as part of the NFS package:

command	function
biod	asynchronous i/o daemon
mountd	server side mount daemon
nfsd	NFS server daemon
rename	renames files
showmount	shows details of remote mounts

TABLE 4. New commands

Biod is a performance aid. Output operations over the network are always synchronous, which slows down large writes by a considerable amount. Multiple copies of *biod* may be run, each of which loops, doing "asynchronous" output over the network on behalf of other processes. Thus if 4 biods are running, 4 write transfers can be performed simultaneously without any user process having to wait for their completion.

6. Future Directions

During the course of this port, we found a number of places in the NFS software where further research might yield gains in performance or functionality. Here is a preliminary list of things we plan to investigate:

1. Vnode locking above the VFS, to prevent local processes from interacting in destructive ways. It will never be possible within the NFS architecture to provide process synchronization over the network, but since inode locking is no longer meaningful above the VFS, some potentially risky interactions have become possible.
2. Adding the 4.2 BSD fast file system as a performance enhancement option.
3. A variety of long file name compression algorithms.
4. Allowing files to be truncated to a specified size.
5. A more generalized transport layer interface within RPC. AT&T's proposed Transport Service Interface[5] is a likely candidate.

7. Acknowledgements

It is in the nature of a porting project that much of the work done is not original. We should like to thank those at Sun who, by producing and maintaining a sound original design, made it possible for us to get the job done on time. In particular, thanks to Rusty Sandberg, Bob Lyon, Steve Isaac, and the Sun NFS consulting group.

Jon Lancaster and his staff made it possible for us to test our software against no less than 14 other NFS ports. The logistic problems he solved in getting 20 computers and 60 people from 15 companies into one room defy description.

Sun Microsystems, Inc. provided us with hardware, office space, and a technically fertile environment.

Excelan, Inc. generously provided Ethernet hardware, software, and support.

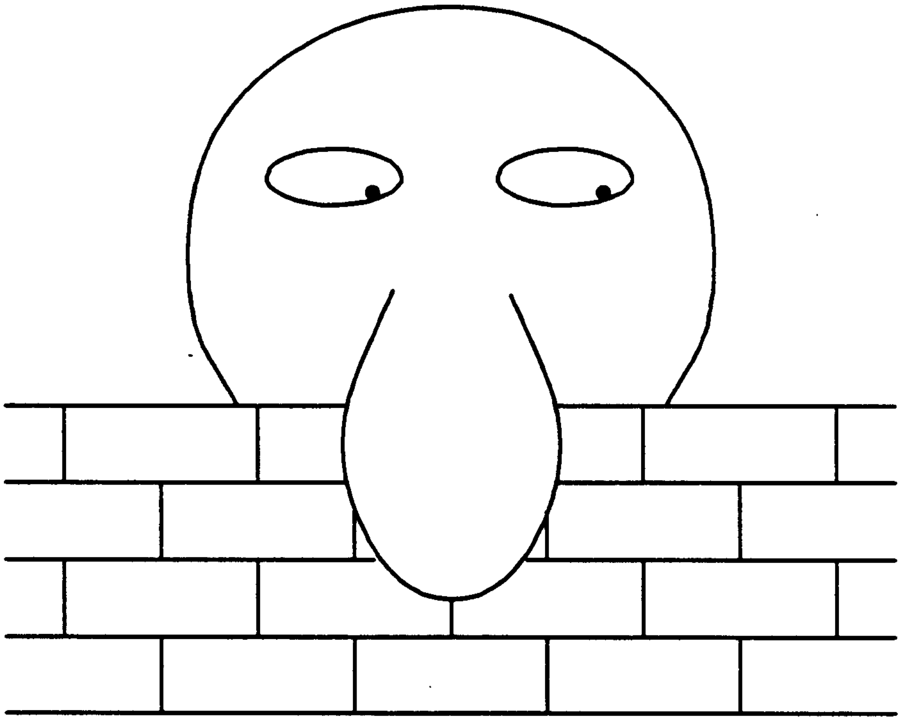
Finally, our thanks go to Daniel Goldman, Ezra Goldman and Sally Rutter.

REFERENCES

1. The System V Interface Definition, AT&T, 1985
2. An Architecture for Multiple File Systems in Sun UNIX, S.R. Kleiman, Sun Microsystems Inc., 1985
3. Design and Implementation of the Sun Network Filesystem, Sandberg et al., Sun Microsystems Inc., 1985
4. Fast Fits, C.J. Stephenson, IBM Sys. Journal
5. Proposal for a UNIX Transport Service Interface Standard, David Olander and Gilbert McGrath, AT&T, 5th September 1985

Fiore Project: Wide Band Area Network

M. Guarducci



"Wot, No paper?"

Fun With UNIX† at Lucasfilm

Michael Hawley

The Droid Works
P.O. Box CS 8180
San Rafael, CA 94912

ABSTRACT

Why do computers and movies mix well? And what does UNIX have to do with it? Before the Lucasfilm Computer Division becomes history we should consider why research there has been so much fun. We claim that computers and rich interfaces – like the arts – are an inherently fruitful mix. Audio programming and music programming are especially exciting since the audio channel is such a powerful and little-used communicator. Furthermore, the rapid rise of synthesizers has only recently made it possible to purchase a rackful of truly *musical* instruments that are computer-controllable. We will briefly describe a simple UNIX/music system that uses Sun workstations to control cheap, store-bought synthesizers, is easy to set up, sounds amazingly good, and is fun to program. The music software is in the public domain.

1. Introduction

> syn.the.sis *pl* syn.the.ses
> Etymology: Gk, fr. *syntithenai* to put together, fr. *syn-* + *tithenai*
> to put, place –more at DO
> 1) a) n, the composition or combination of parts or elements so as to form a whole
> b) n, the production of a substance by the union of elements or
> simpler chemical compounds or by the degradation of a complex compound
> c) n, the combining of often diverse conceptions into a coherent whole;
> *also*: the complex so formed
> 2) a) n, deductive reasoning
> b) n, the dialectic combination of thesis and antithesis into a higher stage of truth

Computers are fun at Lucasfilm: they are married to movies, and that is a beautiful synthesis.

Precise notational tools – UNIX – powerful information engines – like the ASP and PIXAR computers – and an incomparably rich interface – film – make a synergistic mix. Film is an opulent medium, and simulating its complexity can require considerable horsepower and flexibility. UNIX binds powerful tools together simply. Like a good algebra, it can hold up under the weight of heavy and complicated abstractions. Lucasfilm is infamous in certain circles for producing 50,000-line megabyte-shoveling software monsters that take hours or days to compute a few frames of synthetic images or a few minutes of sound; and that notion grates with the crisp minimalism of UNIX, but the two coexist well. Besides – what other operating system would you throw in the path of an onrushing $40,000,000 movie?

† UNIX is a trademark of AT&T Bell Laboratories.

2. The Computer Division: Whence and Whither

Since about 1980, the Computer Division at Lucasfilm has been researching basic aspects of imagery and audio. The charter was open: to computerize any interesting parts of the film-making process. This has included general and far-reaching work in digital audio (under James A. Moorer) and digital imagery (under Ed Catmull and Alvy Ray Smith), as well as more applicative projects, like very high resolution laser scanning and printing of color film, the construction of something like a "word processor" for film and sound editors, and sensory-intensive video games of the future (that run on the home computers of today). The graphics group has produced visual effects of wonder and beauty for major motion pictures (*Star Trek II, Return of the Jedi*, and most recently, *Young Sherlock Holmes*). The audio group has provided sound effects and special processing for sound tracks in *Amadeus* and *Indiana Jones and the Temple of Doom*, among others. Productions are consuming, but they do much to help the technology grow.

Big Numbers†

The brief computer-generated animation *Andre & Wally* was computed intensively over several weeks, using a dozen machines and exercising the latest in rendering algorithms. The complexity of synthetic images astounds: *Andre & Wally* was computed at a resolution of 512×488 pixels; was 1.4 minutes long; had 809 animation controls; and contained 2.9 million polygons per frame (on average). This requires the equivalent of about 1 year of compute time on a VAX 11/780, and 25 man-months, and if computed at film complexity (higher resolution) would require about 17 years of VAX time. The rendering of a "typical" (for Lucasfilm) frame requires about 6-8 Mbytes and 1-2 hours on a VAX; bad cases are more gruesome.

Audio processing is also a labor- and computationally-intensive job. For instance, while 2 or 3 editors can edit the picture for a feature film, a staff of 15 or 20 to deal with all the sound is not uncommon. It has been estimated that the processing power in an average 32-track mixing desk (gain and equalization control, etc) requires about 60 MIPS in digital terms, and to complicate matters, the nature of the processing changes in real time. The size and bandwidth of currently available secondary storage are limiting factors: a 300 megabyte disk pack, formatted in the standard way, can hold about 42 minutes of 50Khz 16-bit monaural audio. A disk transfer rate of about 800 Kbytes per second is required to sustain about 8 channels of audio at the 50Khz rate, and since the mean disk transfer rate is about 980 Kbytes per second, scheduling and buffering algorithms must be shrewd to permit long, continuous transfer and playback. The Lucasfilm sound effects library is currently stored as a wall of ¼-inch audio tape, so auditioning and transfer of effects is a tedious thing. Random access and nice browsing interfaces with pushbutton playback are obviously desirable, but the audio library will require (we estimate) 6 to 10 Gbytes of storage. In the mixing process, considering the different phases of a mix (from spotting to premixing to final mixing), and the multitude of tracks (several tracks for dialog, music, special effects, Foley effects, Darth Vader's voice, etc), as many as 130 reels of sound may go into a complicated scene. This gives some idea of the magnitude of the problem.

New Computers: PIXAR, ASP.

The numerical problems of digital movies are awesome. We have addressed them by designing specialized computers: the PIXAR, an *image computer*; and the ASP, an *audio computer*. Both are 20-40 MIPS microprogrammable devices. The PIXAR is a SIMD machine with a 4-channel instruction (hence the high speed — 4×10 MIPS). The machine was originally intended to solve the ***picture compositing*** problem, in which digital images are matted (combined, synthesized) together to form a single image. Algorithms based on a 4-channel (red, green, blue, and translucency) compositing algebra were prototyped and proven under UNIX, and influenced the hardware design. Often audio and graphics algorithms (such as noise reduction, fast fourier transforms, rendering algorithms, etc) are implemented under UNIX before migrating to

† No Lucasfilm paper would be complete without a few Big Numbers.

microcode. This kind of migration is quite natural since the fast processors are typically driven by 68000-based UNIX systems (like Sun workstations). Other more complex applications divide the work between the ASP or PIXAR and its host, for instance, giving the user a bitmapped display full of controls for a painting system, or windows containing soundfiles that can be edited or arranged in a sound track.

New Companies

As the technology has matured, it has become evident that markets are not limited to the film industry, so Lucasfilm has spun off the Computer Division to form two new companies: PIXAR, which will market the PIXAR image computer and other graphics technology, and The DROID WORKS, which will market computer assistants for film, video, and audio industries.

3. Loud Programs – More about Audio Computing

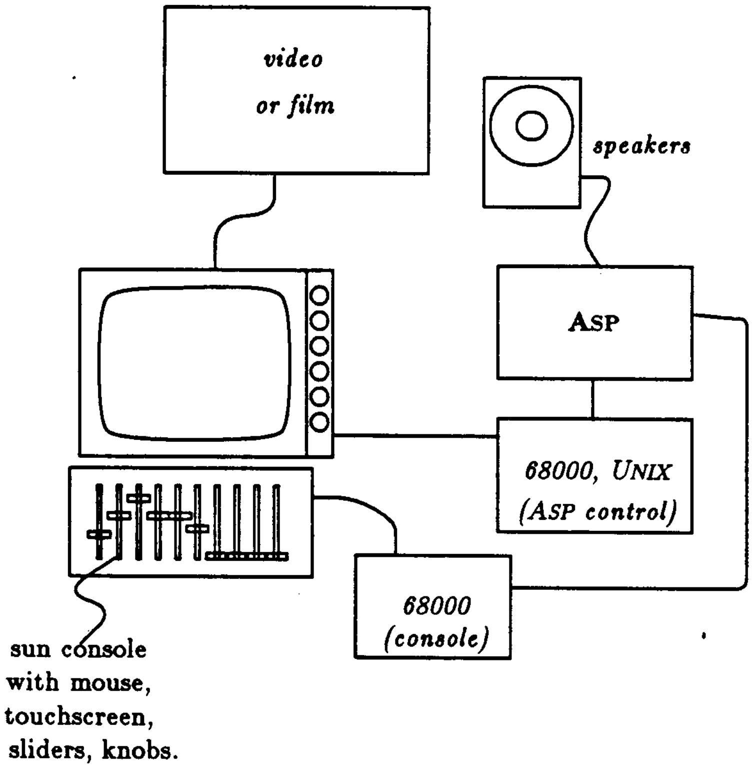
We are beginning to take for granted the availability of graphics in interfaces, but the audio channel, by and large, remains neglected. There are important reasons why most books are printed in black and and white, without distractions like color or sound tracks; and the clacking and beeping of computer terminals is already annoying – what would it be like if obnoxious sound effects like nose crunches, laser blasts and *wurfs*† were gratuitously and unnecessarily thrown in? But sound is a powerful and little-understood communicator. It has to be used to be appreciated. For instance, audio and image interact in curious ways at a fairly low level: viewed with the same sound track, movies seem to sound about 2dB *louder in color* than they do in black and white. There is no scientific explanation for this phenomenon yet, as far as we know. For another thing, sound hits us hard, especially in environments where computers are usually silent. In graphics programming, an off-by-one error typically results in a few relatively harmless smudged or skewed pixels, but this is not even remotely like the astonishment of suddenly shifting the magnitude of audio samples from 60dB to – *woops!* – 120dB: audio goofs routinely blow fuses, exceed the threshold of pain and knock users off their chairs. Looking at "jaggies" (aliasing) is bad, but listening to them is singularly unpleasant. Sound seems closer to emotional response than image, and in relative numerical terms, it is probably easier to compute and evoke powerful feelings with sound than with image. This is perhaps because the production of audio depends critically on *real time* processing‡, on the ordered spacing of events in time. From low-level signal generation to high-level features like the patterns of rhythm and musical phrasing, *temporal spacing* is a critical factor in audio processing, and it affects human perception, emotional response, memory, and learning, in dramatic ways that we have hardly begun to appreciate.

On all levels, audio invites time-critical programming, and demands new notations for expressing such things. To this end, Curtis Abbott has recently developed an interactive language, called CLEO, which has a syntax resembling C and special constructs (*triggers*, *schedules*, *wait* statements, etc.) for dealing with time. At the lowest level, the updating of microcode – plugging and unplugging of software equalizers, etc – is accomplished by maintaining a hardware update queue, which CLEO also deals with.

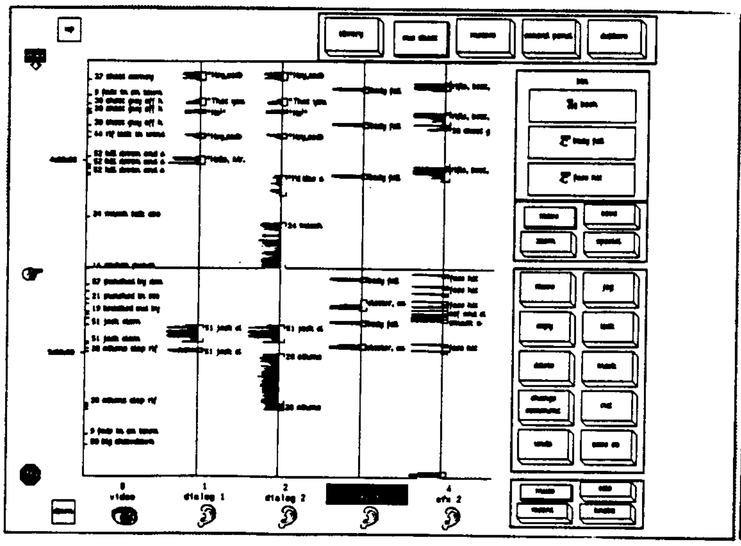
Too, although the audio channel is one-dimensional, parallelism is inherent. Attempting to schedule and synchronize the performance of several musical instruments invites parallel solutions. The parallelism in musical streams is one of several reasons why computerized typesetting of complicated musical scores is still a dream, for example. So, it makes sense for us to divide the film sound problems among a number of processors. One system based on the ASP is called *SoundDroid* (a sound designer's workstation) and it currently uses three processors that cooperate closely:

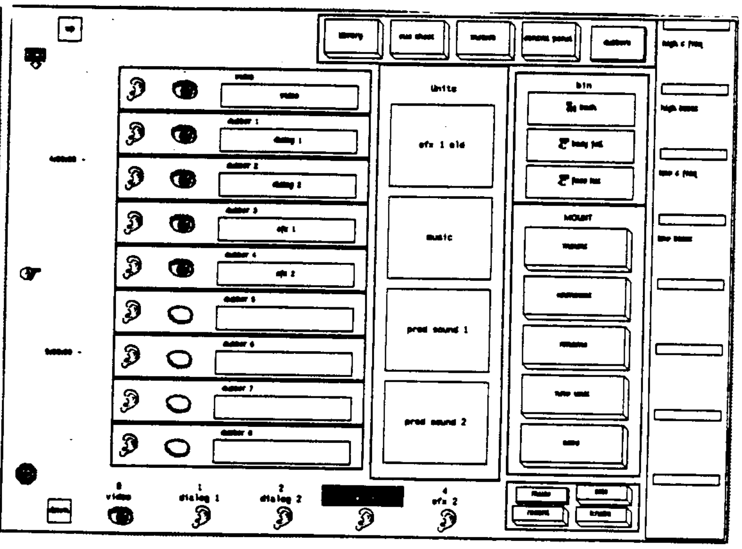
† *wurf* – Hollywood term for a stomach-punch sound effect.

‡ as opposed to "real fast" processing



A Sun workstation with bitmapped display and touchscreen provides the user interface, and communicates with two other processors: one reads the physical knobs and sliders on the console, storing and time-stamping users' gestures and forwarding them to the ASP; the other drives the ASP, managing the loading, plugging and unplugging of microcoded patches, and the scheduling of playback. Also, since the function of the machine is determined by software, it makes sense to build a "soft" graphical console. Mixers mount only the sound tracks they need, and display in their desk only what processing elements are required. The displays the user sees are full of critical information: one window presents sound tracks in parallel (with time running from top to bottom), one shows a screenfull of processing components, another a library of sound effects, etc:



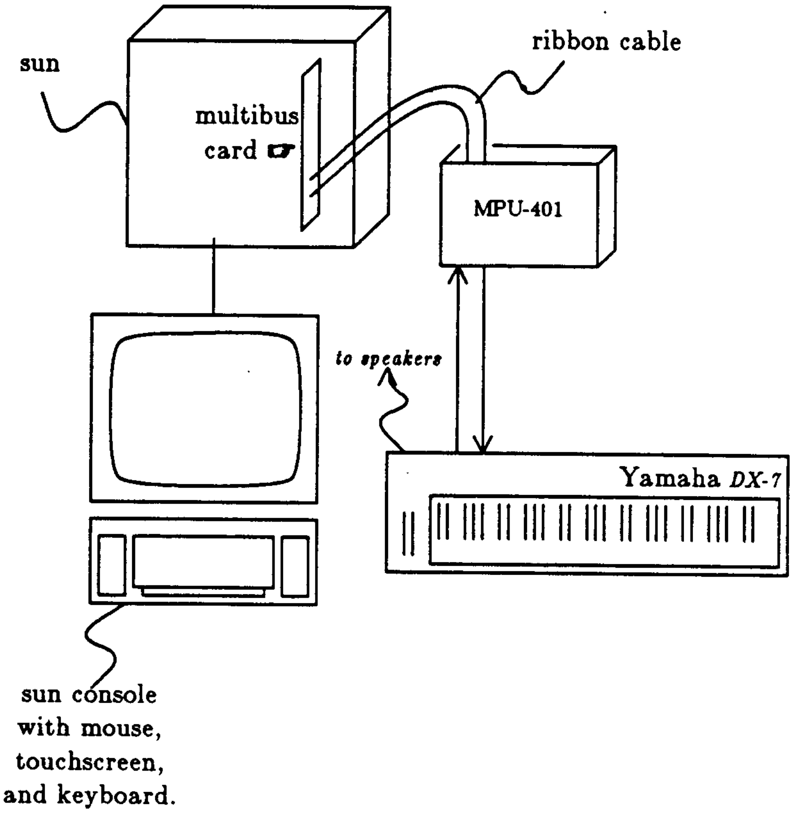


For instance, cutting and pasting of sounds is done by "pressing the glass" once or twice, and the result is displayed and auditioned instantaneously – a far cry from laboriously copying physical magnetic tape and splicing it into a reel at the proper place. Keeping such a system as interactive as possible is a constant challenge.

4. Cheap Music: UNIX and MIDI

The ASP is a vast audio research tool, and unparalleled, especially for signal processing. Still it will be a long time before there is an ASP in every living room – even George Lucas's. Fortunately, the recent advent of MIDI (a communications protocol something like RS232 for controlling synthesizers) brings inexpensive and interesting synthesizers under computer control. For a few thousand dollars – a fraction of the cost of an ASP and competitive with the price of a reasonable piano – it is possible to purchase synthesizers that produce sounds of remarkable complexity and can be easily manipulated by computer.

We use a Sun workstation and a Roland MPU-401 MIDI processor to play, record, and compute music with MIDI synthesizers. Our Sun/MIDI setup is derived from work done by Gareth Loy et al. at the University of San Diego. A typical configuration:



The Roland device is a small, cheap (about $200) MIDI processor. It was designed for use with personal computers and is capable of controlling 16 MIDI channels (*i.e.*, instruments). It typically manages real-time problems like playing and recording. For instance, scores consist of time-tagged lists of events – key on, key off, parameter changes, etc. – and to play a score, an application simply writes data to the device. The MPU-401 buffers and plays score data, interrupting the host when more data is needed. A reasonable driver and higher-level libraries permit programming at a more humane level. Following are a few programs, which, while trivial for computer scientists, inkle at the possibilities.

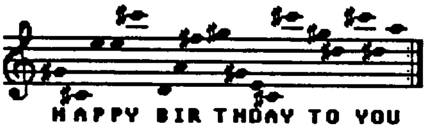
Instant Muzak

The muzak program filters ascii text to music by simply playing ascii values as notes. The basic filter is simple:

```
int MIDI; /* file descriptor of Roland device */

note(n){ /* play pitch n */
      int velocity = 100;
      NoteOn(midi,n,velocity);
      nap(Tempo);  /* sleep for Tempo 60ths of a second */
      NoteOff(midi,n);
}

main(){
      char c;
      MIDI = open("/dev/midi",2);
      if (midi < 0) perror("no MIDI — bleagh!"), exit(1);
      MpuSend(midi, MPU_RESET, 0);
      while ((c=getchar()) != EOF)
            note(c);
      exit(0);
}
```

The actual program is a bit more complex, but not much. It has options to do things like play punctuation characters longer than letters, play louder and faster inside parenthesized groups (so nested loops in C code sound frantic), etc. It also spells out characters while hammering out their pitches on the synthesizer, which adds a lot to the fun. This makes a new kind of musical "word painting" possible. The following makes a cute, even beautiful, tune:

```
hip... hip... hippety hip hap... hap...

happy birthday to you
happy birthday to you
happy birthday dearrrr  aannnnnddddddddyyyy y  y y  y  y y . . .
happy birthday to...    [you] {!!!}
```



Playing kernels and other binaries tends to be tedious and spastic, but alphabetized listings, heavily punctuated dictionary definitions, and numeric files (like octal dumps) are driving and exciting. With English text, letter frequencies give tonal colors to the sound – happy birthday is in c# minor with a bitonal hint of D major, and the similarity between the melody happy (G#-C#-E-E-C#) and ...hday (G#-E-C#-C#) is musically interesting (and serendipitous).

Playing in Parallel

The command

```
play -s a b c
```

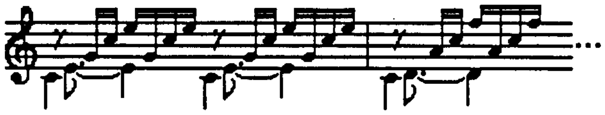
plays MIDI files a, b, and c simultaneously (in parallel). In addition, the play command (and other commands) can accept piped streams as well as filenames, *e.g.*:

```
play -s "|cmd1..." "|cmd2..." ...
```

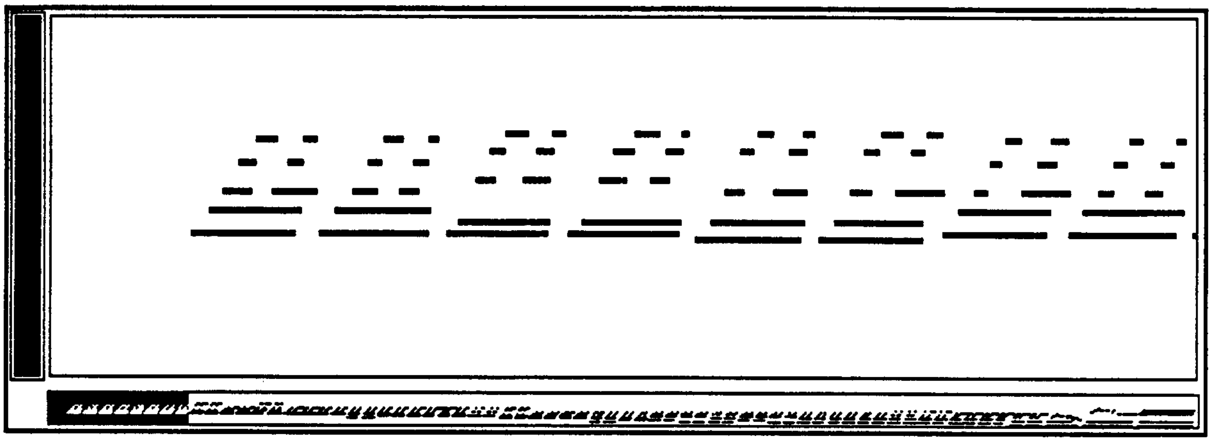
where the notation `"|cmd1..."` means "read from the command `cmd1...`" This allows both files and live processes to provide streams of input. The notation is implemented by a simple pair of routines, `sopen()` and `sclose()`, which combine `fopen/fclose` and `popen/pclose`. `sopen(name,mode)` opens `name` as a pipe, using `popen()`, if `name` begins with a bar `"|"`; otherwise `fopen()` is used. Infix commands can (and often do) execute on remote systems, so processing can be done in parallel. This facility has proven useful in a number of other contexts.

Upside-Down Bach

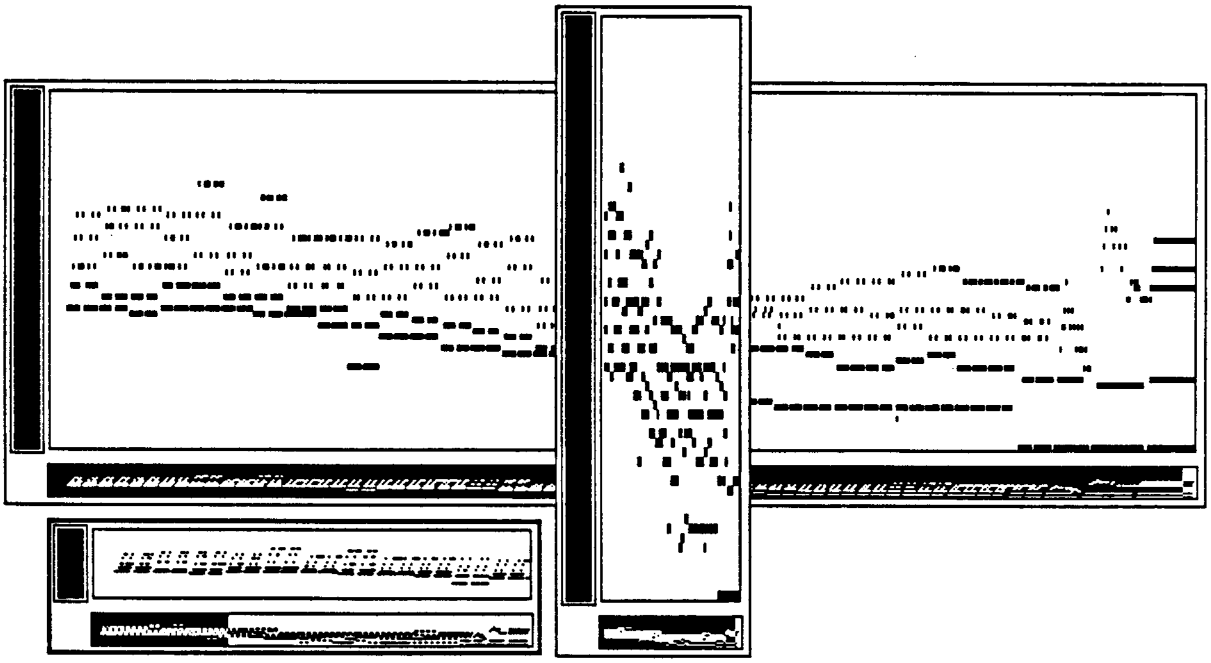
The first prelude from Book I of Bach's *Well Tempered Clavier*



played into a computer looks like this:



This editor displays score data in piano roll format; the scrollbar gives a bird's-eye view of the score, showing the whole piece and the part in view. We can zoom in or zoom out, or by reshaping the frames, look at the music in a variety of new ways:



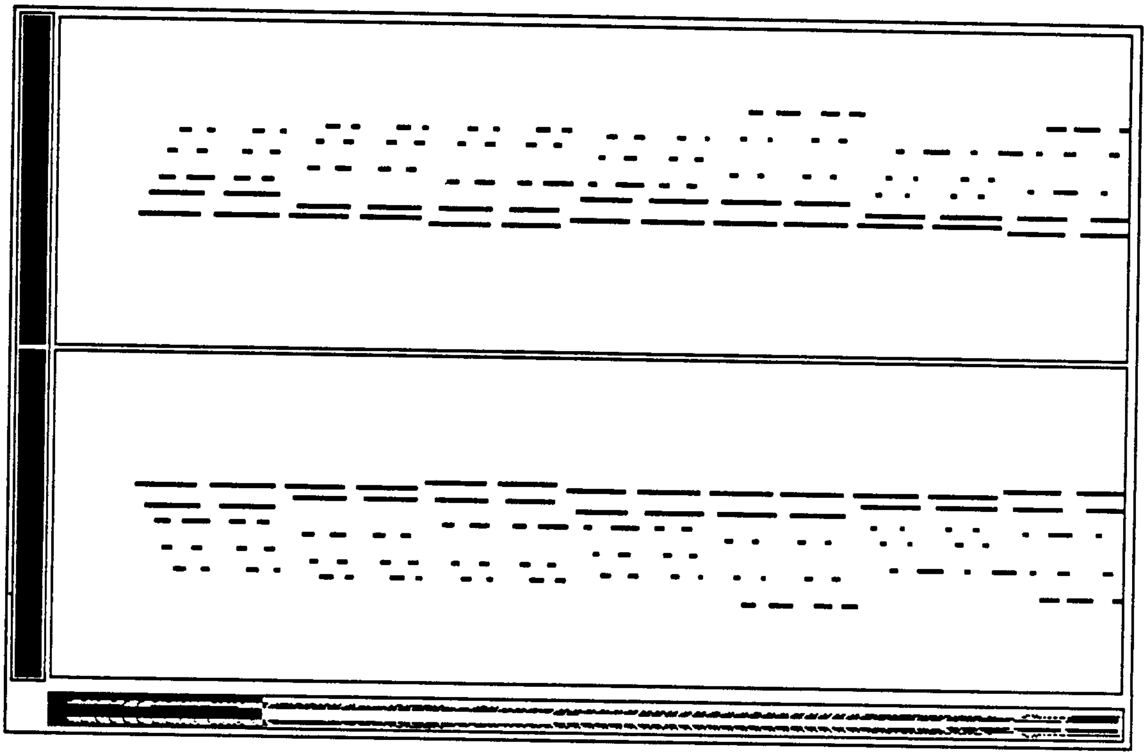
We can invert the piece by mapping pitches, *p*, in the range `a...b` linearly to an output range `A...B`:

$$((B-A)*(p-a))/(b-a) + A$$

This simple filter does that:

```
#include <stdio.h>
#define Alloc(x)   (x *)calloc(1,sizeof(x))
#define md(a,b,c)  ((long)(a)*(long)(b))/(long)(c)
#define transform(v, a,b, A,B) md(B-A, v-a, b-a) + A

between(a,x,b){ return a <= x && x <= b; }

main(){
      MpuCmd *m = Alloc(MpuCmd);
      char p,
            a=dx7_MIN, b=dx7_MAX, /* input range */
            A=dx7_MAX, B=dx7_MIN; /* output range */

      while (GetMpuCmd(stdin,m)) {
            if (IsNote(m) && between(a,p=MpuPitch(m),b))
                  MpuPitch(m) = transform(p, a,b, A,B);
            PutMpuCmd(stdout,m);
      }
}
```

Inverting over the entire keyboard range is like playing on a keyboard with high notes at the bottom and low notes at the top. The rightside-up and upside-down versions together look like this:



which is not visually surprising, but this particular piece has a strange beauty when played upside down. On the other hand, Art Tatum arrangements sound horrible this way, because keyboard and melodic idioms don't invert as well as harmonies – there is more to this class of composition than simply running random material through an informational meat grinder.

Contrapuntal inversion has long been a popular, if academic, compositional device, but full ***harmonic inversion*** hasn't happened before: without a computer, it's just too tedious to rewrite the notes. Inverting harmonies maps major into minor, and vice-versa, and classically grammatical harmonic progressions are transformed into modal ones. For example, a conclusive-sounding I-IV-V-I dominant→tonic major cadence (*e.g.*, C-F-G-C) turned upside down becomes modal and minor (and retrograde) i-v-iv-i (f-c-b^b-f). The result is a music that sounds at once both old (because of the modalisms) and very new at the same time.

2. Conclusions

New technology invites new discovery, and new discovery demands new technology. Beyond this kind of motherhood and apple pie, we are beginning to see the long-awaited renaissance computers were supposed to bring. Research at Lucasfilm points this out: multimedia applications are fun and rewarding, and every computer research group should have easy access to a room full of synthesizers and frame buffers. Concerning music in particular, interesting synthesizers are finally cheap and computer-controllable. Professional quality computer music systems can now be assembled for the cost of a good piano: the technology for it is affordable and need not be restricted to large institutions. There is still no replacement for a Hamburg Steinway, and in some ways it is silly to emulate the sounds and techniques of conventional instruments when so many new things are possible. But there is no replacement for the new technology, either: at last, we have a set of tools we can use to make a synthesis of computers and arts worthwhile.

3. Acknowledgements

Alvy Ray Smith, Rob Cook, Andy Moorer, and Tom Holman provided interesting Big Numbers and other inspirations. Mike Russell made useful suggestions regarding this paper. Gareth Loy *et al.* wrote many of the original MIDI utilities, including the serpentine device driver for the MPU-401. Tim Peierls wrote the piano roll editor shown in the paper; Peter Nye is responsible for the *SoundDroid* application. George Lucas had the foresight to found the Computer Division, and his continued vision has made this kind of work possible.

Adding Commercial Data Communications to UNIX*

Robert A. Heath
NCR Corp.
Columbia, South Carolina

As the UNIX operating system becomes more widespread in small business systems, the need to add commercial data communications becomes important. NCR in its UNIX-based supermicrocomputer, the TOWER, has supplemented the basic data communications tools provided by AT&T with both industry-standard and internationally standard protocols. The resulting product provides interconnectability in three general areas: asynchronous, synchronous, and local area networking. The paper illustrates which protocols are important for a general-purpose small business system and how UNIX fits into hierarchies such as UNIX-to-mainframe and UNIX-to-UNIX networks.

Asynchronous Networking

Asynchronous networking refers to a number of teletype communication schemes using low-cost asynchronous modems through the switched telephone network. The well-known Call UNIX and UUCP are standard UNIX utilities for asynchronous networking as shown in Figure 1. Call UNIX allows a user who is logged in to a local TTY port to dial out through an outgoing port to log into a remote system. This capability is known generically as virtual terminal. Call UNIX is relevant in a business environment because the destination system need not be a UNIX system; it could be a mainframe, a public data network, or an information service.

For reliable batch file transfer, the UNIX-to-UNIX Copy program, UUCP , provides end-user services such as electronic mail, software distribution, remote printer sharing, and store-and-forward capability. These services are actually by-products of its ability to execute a program on a remote machine, giving UUCP open-ended capabilities. It is this open-ended programmability that has given UNIX users a rich set of peer-level services that are only recently being introduced to industry-standard Systems Network Architecture (SNA) in its Logical Unit 6.2. Because UUCP's link protocol is not standard outside of UNIX, it not applicable to UNIX-to-mainframe or PC-to-mainframe interconnections.

*UNIX is a trademark of AT&T Bell Laboratories.

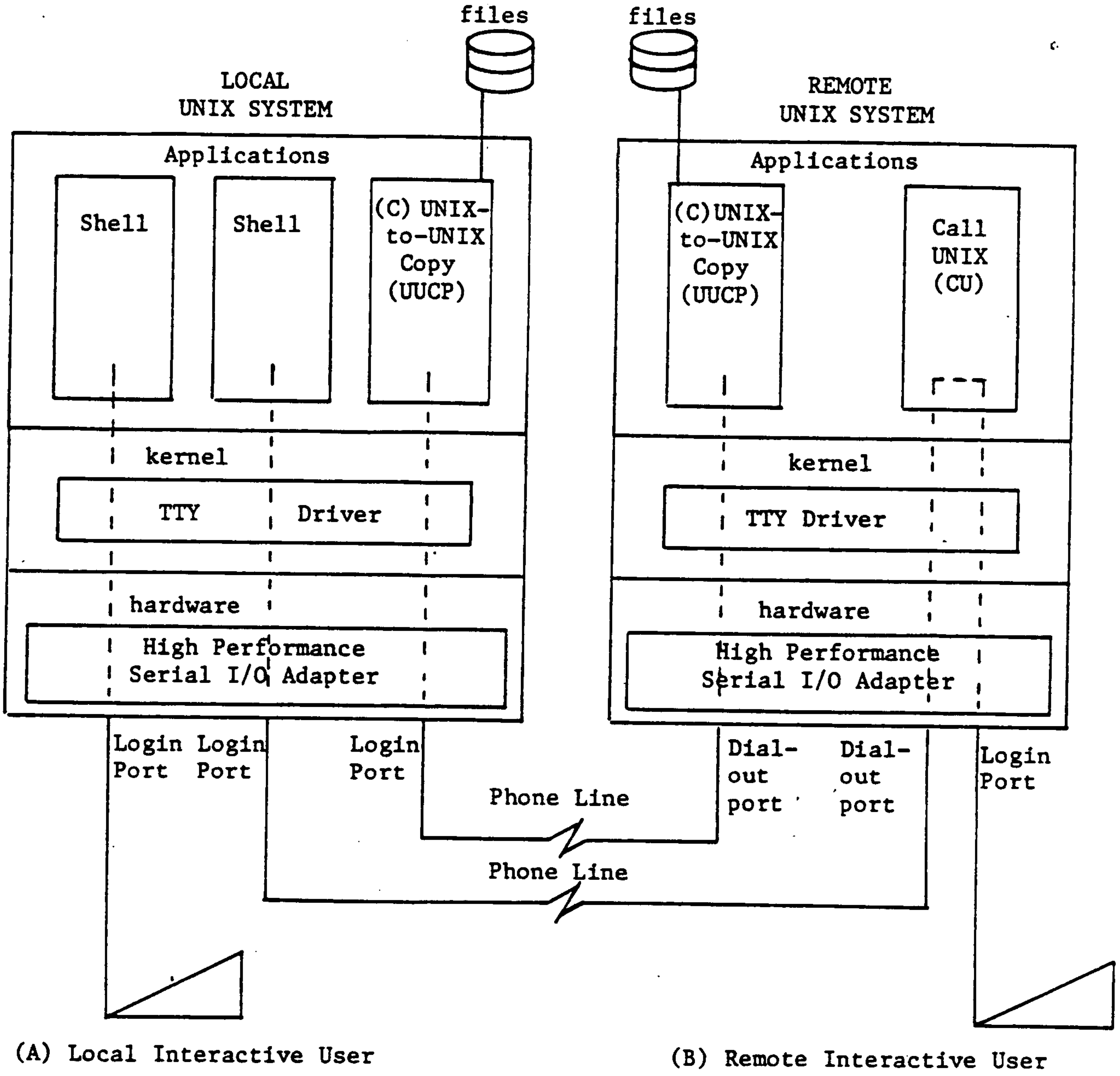


Figure 1. UNIX Asynchronous Networking handles both Interactive and Batch Processing.

(A) A local user interacts with a shell.
(B) A remote user logs into a local system via "Call UNIX."
(C) UUCP transfers files between UNIX Systems.

However, in distributed businesses peer-level communications are commonplace, and where multiple UNIX systems exist, UUCP can fulfill this requirement.

A Front-End Processor for TTY Communications

NCR chose to provide both Call UNIX and UUCP unmodified as part of the standard TOWER software set. However, to complement TTY communications, NCR architected the High-Performance Serial I/O (HPSIO) board, an intelligent, asynchronous adapter which completely transfers the UNIX TTY driver functions away from the main processor. The benefits were immediate: 1) canonical line editing is performed on the adapter and data is presented to the host processor a line at a time, 2) echoes are generated at the adapter level, and 3) data is buffered outside the main memory, avoiding "high water" conditions in which critical buffer availability leads to data loss. This adapter allows large UUCP networks to be set up in-house, using low-cost direct TTY connections running at 9600 bits per second.

Synchronous Networking Functions in Standard UNIX

Synchronous networking encompasses character-oriented and bit-oriented protocols for modern high-speed communications over leased or dial-up phone lines. Though synchronous communications are not new to UNIX, UNIX is still deficient in this area. UNIX System III introduced the Virtual Protocol Machine, a software innovation for offloading synchronous protocols to an intelligent communications adapter. The VPM offered an interpretive language similar to `C' for implementing data link control protocols. Its most well-known application is the RJE package which provides industry-standard multileaving remote job entry. RJE offers a commercial grade of terminal-to-mainframe communications, but it is outdated by SNA techniques. UNIX System V introduced Bell X.25, also based on the VPM, which featured a programmatic interface and served as an alternate vehicle for UUCP. The implementation was not adequate for general X.25 networking because it featured only permanent virtual circuits, omitting the popular switched virtual circuits.

Despite these standard offerings, further packages must be added to upgrade a UNIX-based product in the area of UNIX-to-host and peer-to-peer synchronous networking for a business environment.

What Synchronous Protocols Does Business Require ?

NCR decided to include industry-standard protocols early in the TOWER's product history to interconnect with customer mainframes. The initial bisynchronous offerings were IBM 2780/3780 and IBM 3270. The prevalent SNA terminal node, PU Type 2, was also targeted, along with its associated LU type 1 for remote job entry and LU types 2 and 3 for 3270 display and printer emulation. Later X.25 networking, based on the CCITT international standard, was added for peer-to-peer and UNIX-to-host connection through a packet-switched data network.

Hardware for Synchronous Communications

TOWER synchronous communication is rooted in its Multiprotocol Communications Controller, the MPCC, shown in Figure 2. Like the VPM, this intelligent adapter's firmware offloads the time-critical framing of block-oriented protocols such as bisync, SDLC, and HDLC from the M68010/20 main processor. Unlike the VPM, protocol handshaking decisions are relegated to software drivers within the kernel. Two exceptions are repetitive functions like automatic polling and responding to polls, which would otherwise degrade performance due to their continuous demands on the processor. The main processor generates control programs which reference data buffers in the shared memory area. These low-level programs direct the adapter to send frames, receive frames, or control modem signals, leaving most of the actual protocol logic to the driver.

The adapter is unbuffered, depending on shared memory within the TOWER's main memory. Since the TOWER's main memory is battery-backed, buffer data is preserved across power failures. Like other TOWER peripherals the MPCC adheres to the Multibus* standard. NCR chose to design the adapter itself, rather than seeking an outside vendor, to meet its internal standards for reliability and serviceability.

2780/3780 Bisync Emulation

The first TOWER networking package added was 2780/3780 bisync, which opened doors to peer-to-peer and UNIX-to-mainframe interconnection. (See Figure 3.) Because it can be used either for remote job entry to a host or for peer-level file transfer, 2780/3780 is the most generally interconnectable synchronous

*Multibus is a trademark of Intel Corporation.

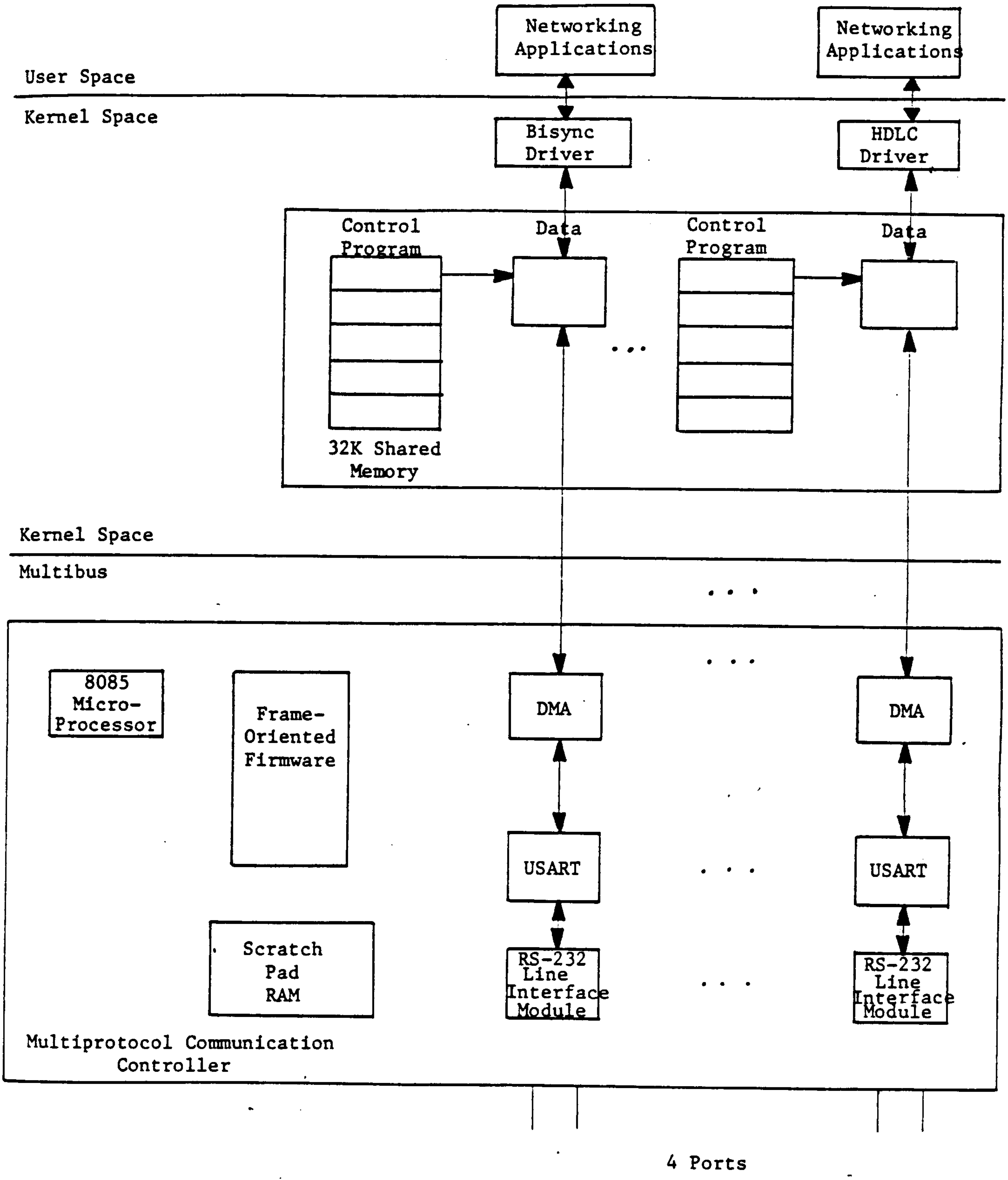


Figure 2. The Multiprotocol Communication Controller offloads time-critical framing for Block-Oriented protocols.

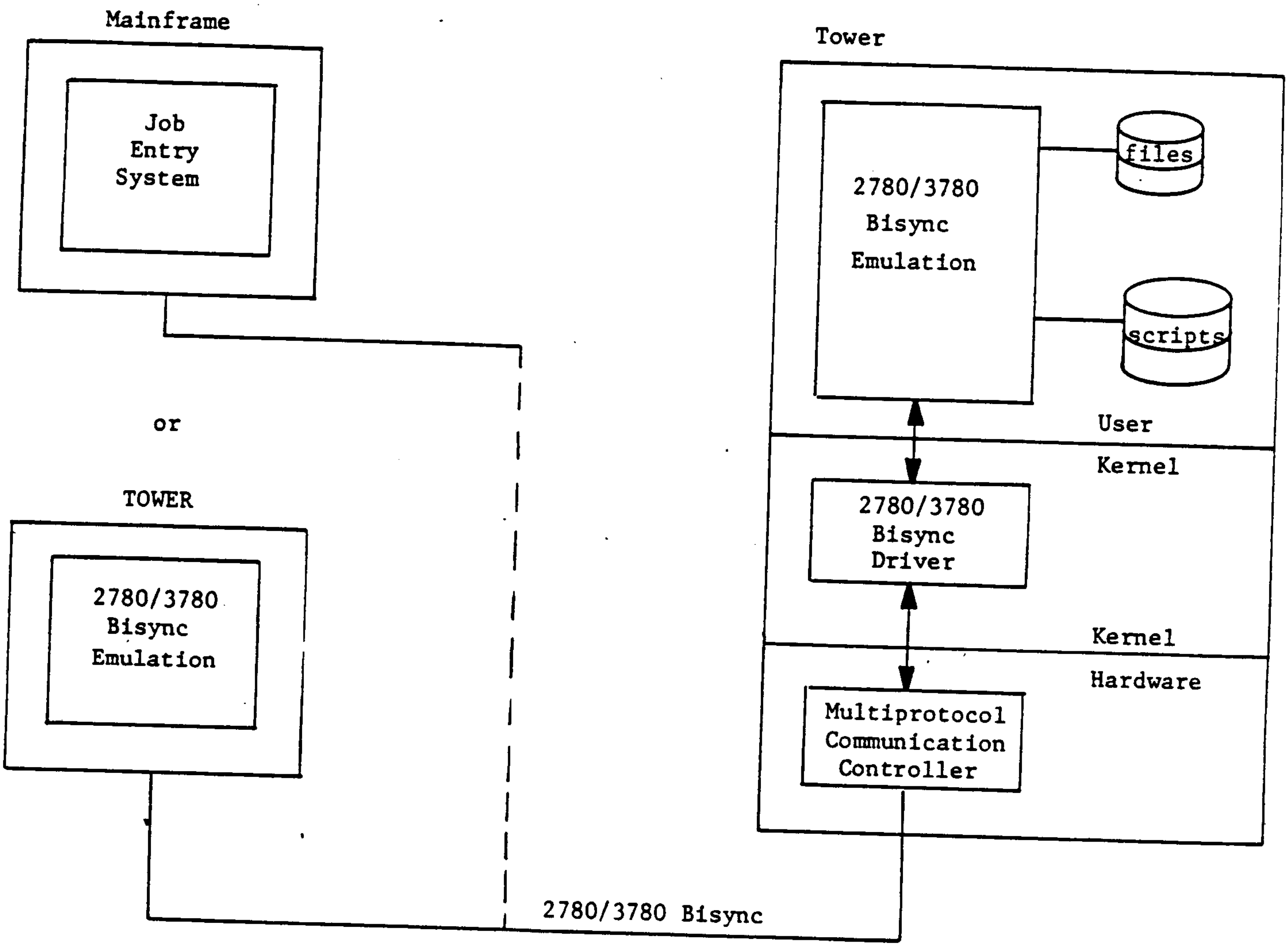


Figure 3. 2780/3780 Bisync provides both UNIX-to-Mainframe or UNIX-to-UNIX Interconnection.

protocol among medium-sized processors.

Within UNIX this simple package consists of user-invoked application, a kernel-based bisync driver, and the Multiprotocol Communications Controller. The application is structured as an interactive UNIX command which works on a one-for-one basis with a driver port. The application accepts high-level commands which send from or receive into a given file. The application level performs less time-critical functions such as packing records of UNIX data into bisync blocks, file handling, and data compression. The driver interface employs the standard UNIX character device primitives open, close, read, write, and ioctl. Though the bisync driver is considered a character device, it transfers data across the interface as entire blocks rather than one character at a time.

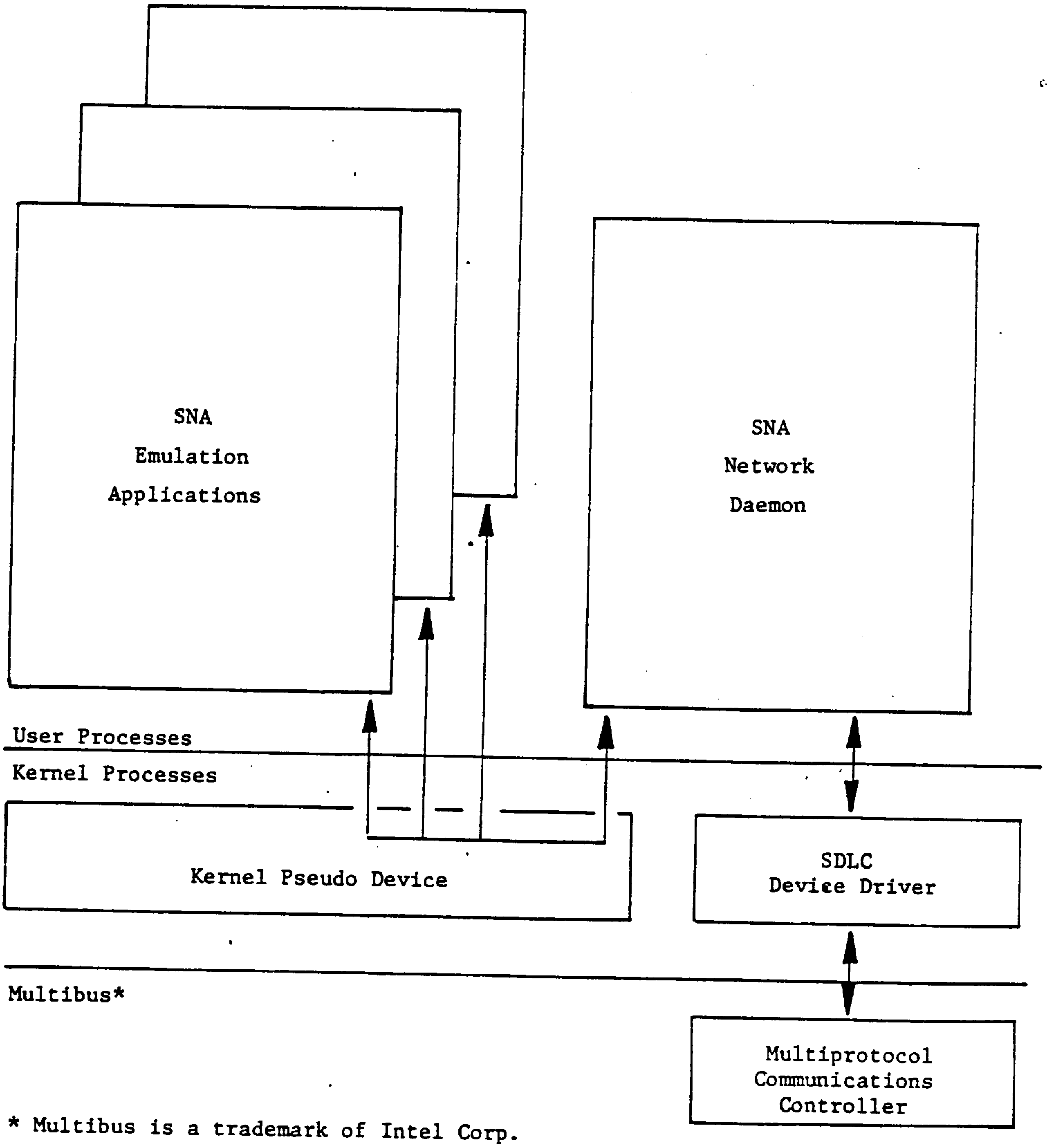
Driver performance is time-critical at the message level. It copies data between the application and kernel buffers, translates between user data and bisync blocks, and performs acknowledgements and retries.

SNA Networking

NCR forecast that many customers would prefer to interconnect their TOWERs with their IBM hosts in SNA networks. Adding SNA to UNIX presented some special problems due to the network architecture's characteristic multiple layers and multiplexed data streams. Other problems in structuring the networking system were that some SNA entities, namely the Physical Unit, were required to be active whenever the system came on line. Meanwhile the user end-points, or Logical Units (LU's), became active or inactive as users entered and exited their different emulations.

The solution was to design a pseudo-device in the kernel to bridge the path from the independent LU's to the lower-level network layers. This is shown in Figure 4. Implemented as a background process, the network layers continuously cycle, keeping the link active and responding to SNA requests from the host. The pseudo-device buffers the multiple SNA applications from the SNA demon, presenting each application with its own device, named appropriately, an LU. This approach saves kernel memory, which by nature is always resident, because the majority of the SNA logic is relocated to the task level.

At its lower-order interface, the network daemon



* Multibus is a trademark of Intel Corp.

Figure 4. Pseudo-device bridges SNA layers in horizontal arrangement within Unix. Data flow actually weaves in and out of kernel when crossing SNA layers.

interacts with a Synchronous Data Link Control (SDLC) driver within the kernel. This driver issues and compares sequence numbers, acknowledges frames from the host, and queues data to and from the network application.

3270 Display Emulation in SNA

IBM 3270 interactive communications have become a de facto industry standard. The original product consists of a cluster controller, intelligent displays, and printers. On UNIX common video display terminals play the role of 3270 displays. (See Figure 5.) An SNA 3270 display emulation program draws on UNIX's termcap library of terminal capabilities to intermix various terminals types. This approach allows users to create their own termcap entries when a new terminal type is introduced. Because video display terminals seldom duplicate the 3270 key set, the application occasionally maps multiple terminal keystrokes to match the 3270 keyboard set. Since the emulation is structured as a user-loadable application, operators freely enter and exit from this role without permanently dedicating the workstation as a 3270 display.

3270 Printer Emulation in SNA

A 3270 Printer Application cooperates with the screen-oriented 3270 Display Application. This program cycles as a background task, awaiting printouts from the host. Using the UNIX line printer spooler, lpr , it consolidates printouts arriving from the host with listings originating locally. Because printing through lpr is spooled to disk, the Printer Emulation always appears ready to accept another printout from the host.

SNA Remote Job Entry

For bulk data transfer between a mainframe and smaller systems, IBM 3770 remote job entry is an industry standard. Since a form of remote job entry, RJE, already existed on UNIX, NCR chose to convert RJE from a package based on multileaving into one based on SNA. The new subsystem, known as SNA RJE, preserves the familiar send and gather commands for their ability to merge job decks from component files. SNA RJE spools user jobs bound for the host and alerts users on returning printouts via mail or write commands. The UNIX file system allows the user to create job decks as files rather than as punched cards, for which 3770 was designed. (See Figure 5.) Similarly, returning printouts and

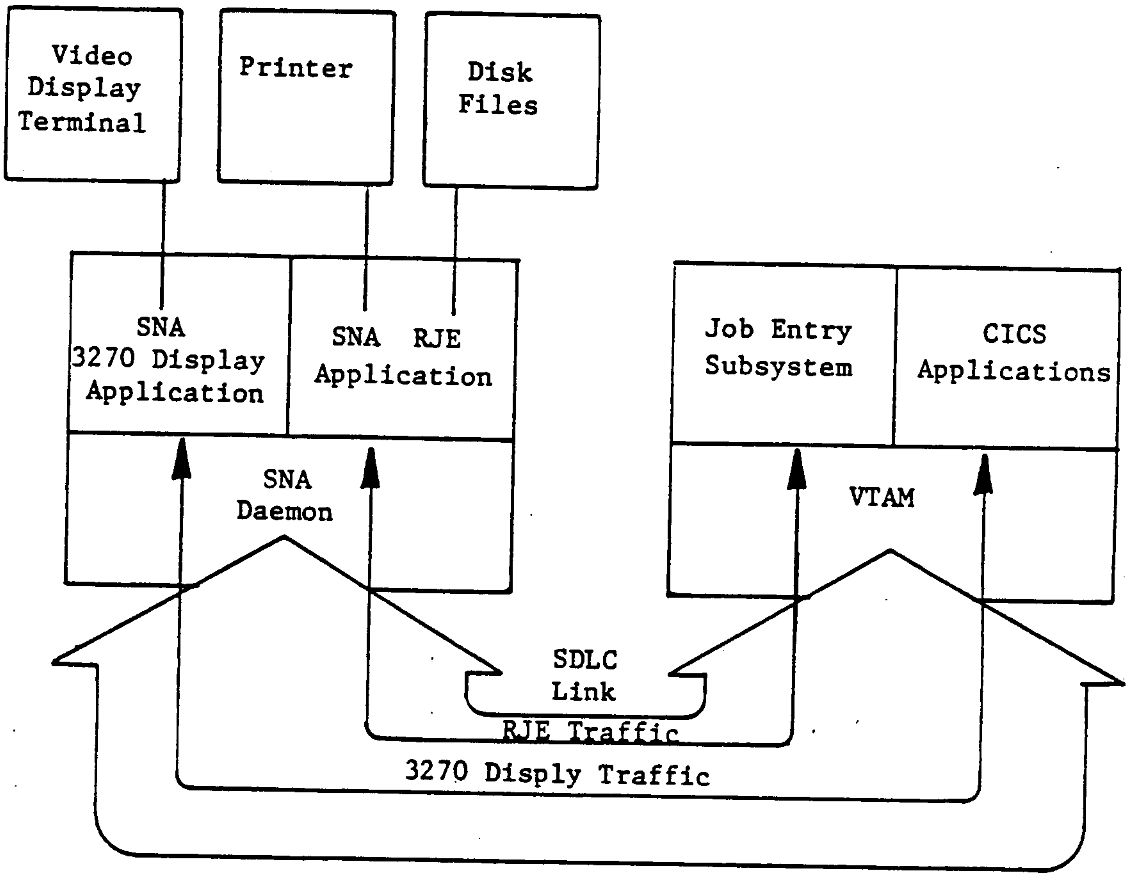


Figure 5: SNA RJE Applications reuse UNIX line printer facilities while 3270 Display Applications reuse UNIX Video Display software.

punch decks become files, allowing the user more flexibility in their disposition.

Under SNA RJE any UNIX workstation can emulate the console of a 3770 terminal. Thus a user can query the host on the status of his job within the host itself. More privileged administrators can issue commands the host's job entry subsystem to alter handling of the job.

X.25 Networking

Adding X.25 to UNIX shared many of the same problems that SNA did because of X.25's characteristic multiple layers and multiplexed data streams. Similarly the user end-points, or network addresses, could became active or inactive as users entered and exited their different applications.

The solution was to reuse the kernel pseudo-device approach to connect independent user applications with the X.25 packet layer just as was done with SNA. (See Figure 4.) The X.25 layer is structured as a background process which keeps the link active and responds to connection requests from remote end-users when the end-points themselves are inactive. The pseudo-device presents each application with its own device, which is the equivalent of a network address. The network process interacts with the HDLC driver, which converts the packets into the frames of X.25 Link Access Protocol--Balanced (LAP-B). The same HDLC driver, configured for different roles, is used for both SDLC and X.25 LAP-B. The term HDLC implies a superset of both protocols.

Combined SNA and X.25

As the popularity of X.25 increased within the early 1980's, IBM announced its plans for X.25 within the framework of SNA. Instead of announcing new end-products tailored to X.25 use, the strategy defined how existing SNA products could be adapted to X.25 packet-switching networks. IBM added a new protocol, Qualified Logical Link Control (QLLC), above the packet layer for mapping of SNA data streams onto X.25.

To reuse its 3270 SNA and SNA RJE emulations over X.25, NCR consolidated its SNA networking software with its X.25 networking software. The resulting SNA/X.25 background process merges the SNA data stream from its emulations, using the QLLC protocol to create X.25 packets. As with the original X.25 product, the background process interfaces to the HDLC driver, which transforms the packets into X.25 LAP-B frames. Since the 3270 SNA and SNA RJE emulations could not tell whether they were being transported over an SDLC link or an X.25 network, it was a good example of how entire levels can be substituted within multilayered architectures.

3270 Bisync Emulation

Figure 6 shows the software architecture of 3270 Bisync within UNIX. The package is structured as a display emulation and a printer emulation which interact with a bisync driver within the kernel. Since the networking software required for 3270 bisync is less sophisticated than its SNA counterpart, the logic can reside totally within the kernel. The driver software replies to specific polls and selections, transforms user data and 3270 protocol blocks, and undertakes recovery and acknowledgments. Most important, it multiplexes the various 3270 devices onto a single line, allowing simultaneous conversations from different workstations. More time-critical duties such as character buffering, checksum generation, and block framing are left to the adapter firmware.

From the user's point of view, the 3270 display emulations for SNA and bisync are nearly identical. Like SNA 3270, the 3270 Bisync emulation uses termcap for terminal-independent screen control. In a similar fashion, it maps an ordinary video display terminal keyboard into the richer, 3270 keyboard set. A special "hot key" sequence is defined, allowing the user to exit to a shell temporarily for UNIX processing then to return to the original 3270 screen image. Operators freely enter and exit the display emulation since it is just another application.

Local Area Networking

NCR added two levels of local area networking to UNIX. TOWERNET provides the basic file transfer and virtual terminal services for interworking multiple, closely-connected systems. Distributed Resource Sharing (DRS) extends TOWERNET, not only creating a distributed file system but also allowing remote devices to be directly accessed.

As with other communications, TOWERNET is structured within server tasks, kernel drivers, and a front-end processor. It is an implementation of the Xerox Network Systems (XNS) Internet Transport Protocols that communicate on an Ethernet CSMA/CD local area network. The XNS protocols themselves are offloaded onto an intelligent, programmable adapter which provides the buffering and rapid response required to support the

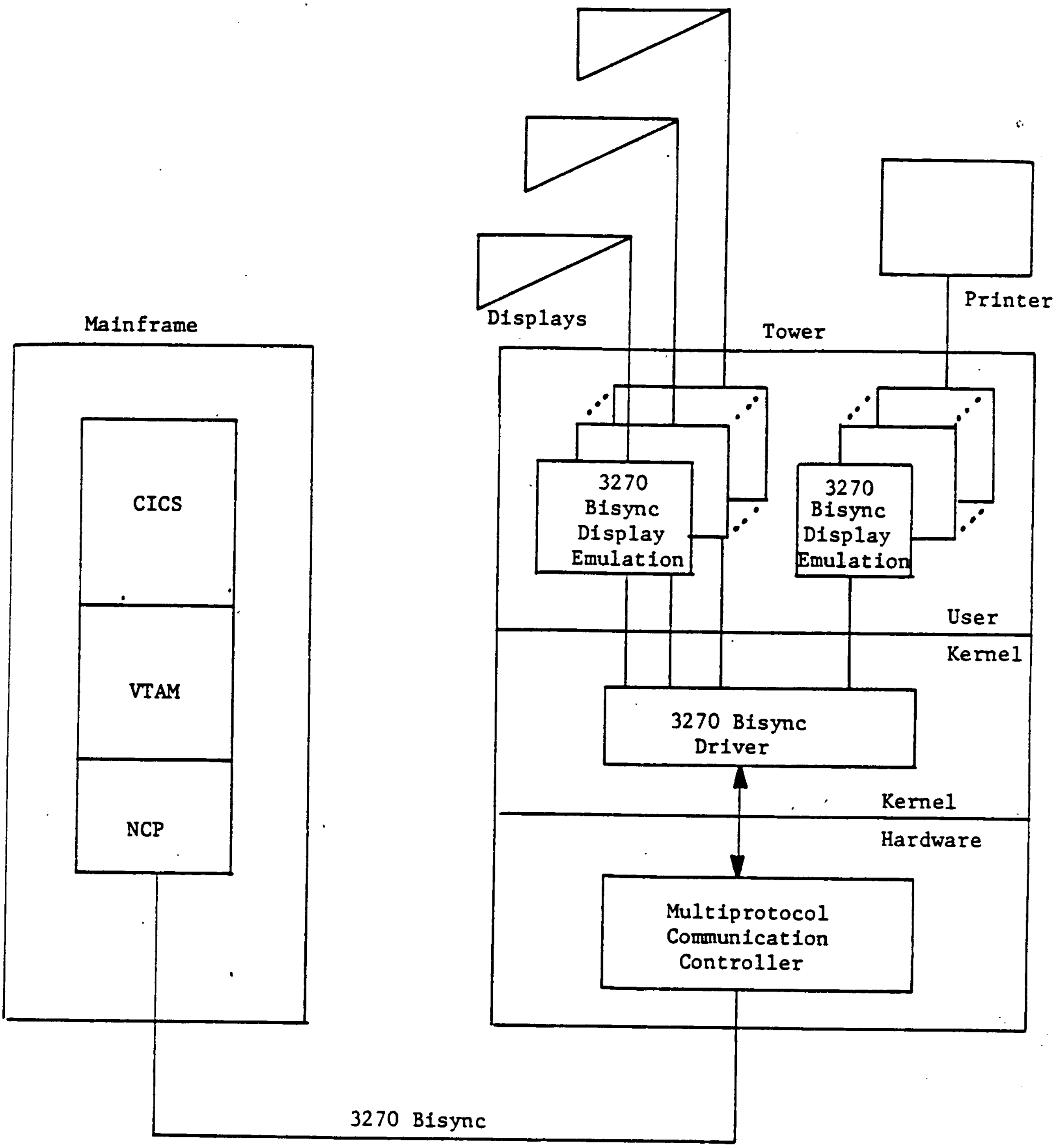


Figure 6. 3270 Bisync Multiplexes both display and printer data on a single line.

distributed file system of DRS. Most of TOWERNET is structured within the kernel to keep response time low. Server processes running as daemons do the following: coordinate the time of day within the network, manage a directory of machine names, and handle remote file access. With any other system in the network, users can send a file, receive a file, log in, or execute a program. These services are subject to security restrictions protected by password.

Distributed Resource Sharing, shown in Figure 7, allows the file systems of remote machines to be mounted within local directories as if they were ordinary disk files. Files on remote machines can be "linked" to directories on the local machine. To keep file access transparent to the user, the DRS interface is composed of ordinary UNIX system calls such as read, write, stat, mount, unmount, etc. Since remote devices are transparently accessed from local machines, network gateways can be created without explicitly adding code for gateway management. An example would be a local user's executing Call UNIX with a TTY device connected to a modem port on a remote machine. Sharing the device across DRS is no different from using a device on the system itself.

Configuration and Management

A great improvement that NCR has made over standard UNIX utilities is in the area of package management. Most TOWER networking packages are installable options. A special System Administrator login, an alter-ego of root, allows all optional packages to be uniformly installed or removed through a menu interface. Once installed, packages are accessed with their own management login for configuring and updating the package. Rather than forcing on the user to edit the package's administrative files by hand, as UUCP does, a menu-driven interface is provided with on-line help screens.

In-Service Diagnostics

A separate login for runtime diagnostics is provided. This separate login is provided for the customer or field analyst to exercise built-in diagnostics. When a communication device is out of service, the analyst can run loopback and pattern tests to exercise the adapter's basic functions. While the device is still on-line, the analyst can run non-intrusive tests such as gathering statistics and tracing line activity to check network quality or protocol compatibility. These additional diagnostic features are an index of quality in commercial data communications.

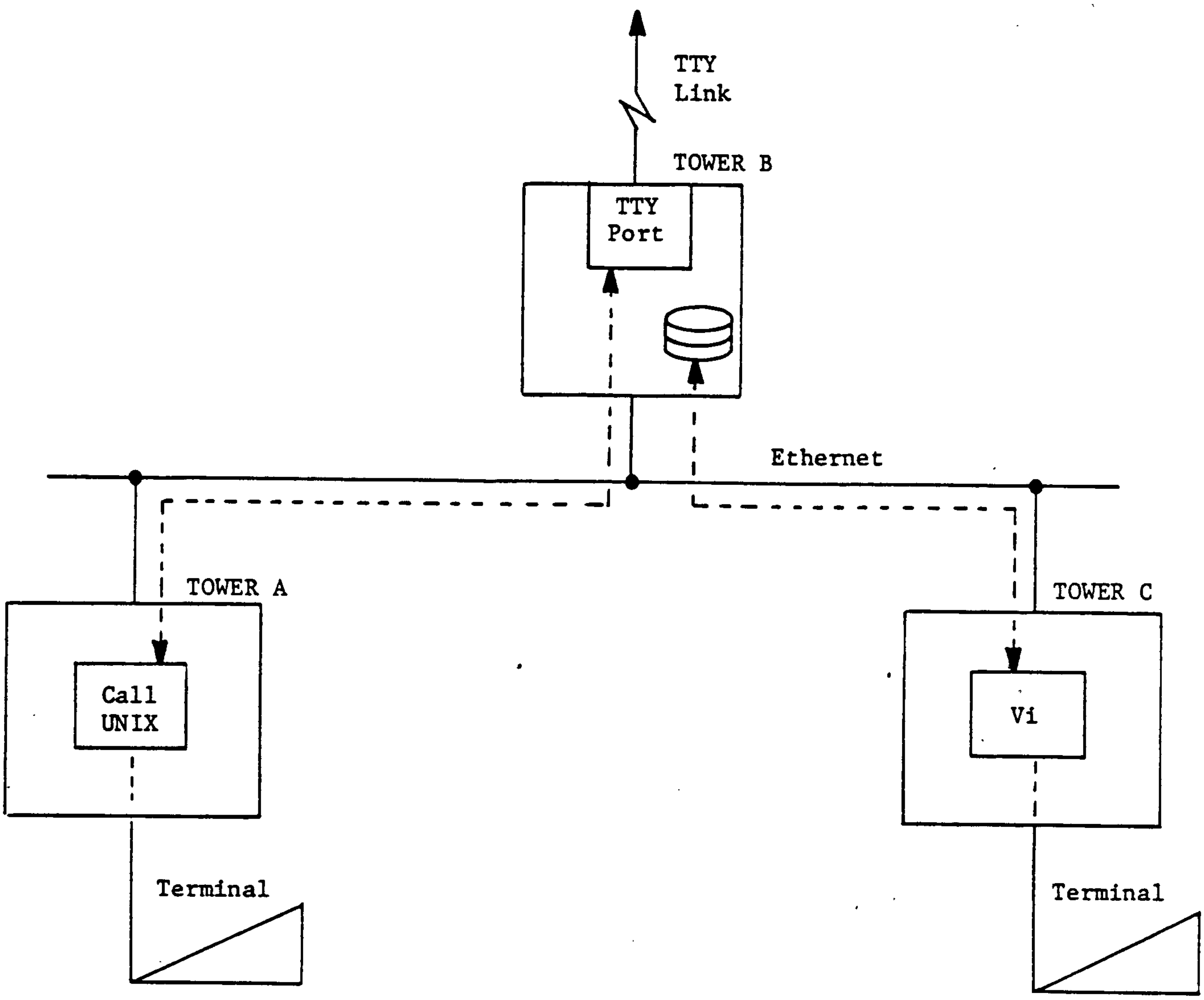


Figure 7. Distributed Resource Sharing

1) A user on Tower "A" accesses a device on Tower"B".

2) A user on Tower "C" edits a file on Tower"B".

Conclusion

Some of the standard UNIX networking utilities are reusable in a commercial setting; others can be adapted to a commercial setting. Generally, more synchronous protocols and local area networking schemes must be added to offer a well-rounded product. Though adding more protocols and end-user services is the foremost concern, these must supported by maintenance and service tools as well.

Workstations: The Next 4 Years

Bill Joy

Sun Microsystems, Inc.

In 1982 at the Paris EUUG Meeting I gave a talk about UNIX Enhancements in 4.2BSD and the emerging workstation environment. That talk foresaw environments of diskless nodes on local networks with file and compute servers. We prototyped that environment at Berkeley using VAX 11/750's.

Our goals were the same as the CMU Spice project: we wanted low-cost and high-performance workstation nodes on our network with high-resolution displays. We wished to emphasize the use of standards and open systems. I described the Sun Workstation, which met the goals, and compared it to the BLIT and the VAX 11/750.

Four years later, UNIX and workstations are major forces in the technical marketplace, and the architecture described in the 1982 talk is widely supported. Early vendors of proprietary operating systems on workstations (Apollo, Perq, etc.) have either switched to UNIX or become significantly less competitive.

1. The first four years

The years 1982-1985 may be considered to be the first four years of workstation computing. In these years, the 680x0 family has dominated the workstation marketplace as well as the price/performance sensitive microcomputer UNIX market. Ethernet and TCP/IP have also come to dominate. Graphics on workstations today is done by host software on entry-level systems, or by special purpose pipelined or bitmapped hardware on high-end workstations.

UNIX applications software is relatively little changed from 1982. In the networking area, Remote Procedure Call (RPC) has emerged as a technique for building distributed applications, and most vendors have provided bitmapped window interfaces based on rasterop-vector-polygon type graphics standards. No single window-system standard has emerged, nor has any radical improvement in the applications set taken place.

2. The next four years

In the next four years we foresee radical performance advances in workstations. Individual workstations will have performance in the 10-30 mips range, with multi-megaflop floating point performance. This will bring many hefty applications and use of new technologies such as AI to the desktop. RISC technology will be the driving force behind the advance of workstation processor performance.

In the graphics area, we expect to see higher-level software emerge. Presentation graphics oriented primitives, such as PostScript, should replace bitblt as the preferred way of interfacing to the screen and output devices. Curves and shaded surfaces should replace lines and polygons as the way of describing 3d objects. High-speed fiber-optic networks and the new ISO protocols should replace TCP/IP.

A new applications environment for UNIX, which is more "macintosh-like" should emerge. A tool writing environment similar to Sun's SunView provides support for writing new display-oriented applications, but it will be several years before any such environment comes into widespread use.

UNIX software developers should see an evolution from a set of modular tools based on C to an integrated environment based on C++. Support for incremental compilation, shared libraries and concurrency will aid programming in the small; higher level tools for configuration control, release management and project administration will replace antique tools such as SCCS.

3. The Next Os: UNIX/1990?

I expect that by 1990 a new operating system base which will supplant UNIX will emerge. This operating system would run all UNIX applications, but would have concepts more suitable for the computing environment of the 1990's. I will speculate on the shape of the 1990 computing environment and the form such a new system might take.

Computer Music under Unix Eighth Edition

T. J. Killian

AT&T Bell Laboratories
Murray Hill, New Jersey

ABSTRACT

We describe an evolving computer music system which draws upon many of the novel facilities of the 8th edition as well as the standard repertoire of Unix tools. The Teletype 5620 bitmap display serves both as the user's terminal and real-time controller. The *mux* window system is used to download a MIDI interface driver which services other windows (by direct code sharing) and host processes (which write on the driver's control *stream).* We presently have two MIDI-compatible instruments, a Yamaha DX7 and TX816.

Window programs include a piano-roll style score facility and a virtual keyboard. Host programs include a music compiler, *m*, which converts an ASCII score notation into MIDI events; it is based on *lex* and *yacc*, making it very easy to develop in response to user needs. There is also a variety of filters which perform simple transformations (e.g., time and pitch translation) on MIDI files. The latter are ASCII, so that, for example, output from the DX7 keyboard can be translated into M notation with an *awk* script, and other Unix text filters (especially *sed* and *sort)* and C programs are useful as well.

1. Introduction - the MIDI standard

Computer music applications have taken off in recent years, largely due to the introduction of the MIDI (Musical Instrument Digital Interface) standard [1]. MIDI has made it possible, with very modest hardware, to interface a computer to synthesizers and other equipment, with broad compatibility among manufacturers. A full discussion of the MIDI standard is out of place here; we will simply give some of its essential characteristics.

MIDI uses an asynchronous serial protocol at 31.25 Kbaud to transmit 8-bit data bytes over a 5 mA current loop; thus, to the programmer, a MIDI device looks like an ordinary RS232 line. *Status* and *data* bytes are distinguished by the high-order bit being set or clear, respectively. A status byte encodes a *function*, and (usually) a MIDI *channel number* in the range 1 to 16. Bytes are grouped logically into *messages* which consist of a status byte and (except as noted below) 1 or 2 data bytes. For example, the 3-byte message (0x90, 0x3c, 0x40) says, "On channel 1, turn on note number 60 (middle C) with a volume of 64 (mf)." In most cases the size of a message is determined by its status byte, so the latter may be omitted if it is the same as the status byte most recently transmitted.

The messages most commonly used in performance, such as those which select a synthesizer's pre-programmed voices, or turn notes on and off, are fixed by the MIDI standard. An escape mechanism is provided to allow control sequences peculiar to the equipment of a given manufacturer. This *system exclusive* message has the form (0xf0, *ident*, *data*, ..., 0xf7); *ident* is a manufacturer's code assigned by the standards committee; the *data* is arbitrary (and can be of any length).

Messages are intended to be acted upon in real time by the receiver whose MIDI channel number matches that of the message. A MIDI performance thus consists of (mostly) note-on and note-off messages emitted by the "performer" with proper timing.

There are several problems with MIDI, the most serious of which is probably its limited bandwidth. Chords consisting of, say, ten or twelve notes become noticeably arpeggiated, and changes in timbre which might require a long system exclusive message cannot be fitted into fast passages. MIDI is also rather tightly bound to the twelve-tone Western scale; it is possible to work around this, again at the expense of bandwidth. Nevertheless, it still allows a rich variety of possibilities, some of which we shall explore.

2. The MIDI device driver

Mark Kahrs built a MIDI interface board which plugs into the parallel I/O port of the Teletype 5620 bitmap terminal. The 5620 has a 32-bit microprocessor with 1 Mb of memory, and runs the *mux* window system which has been described elsewhere [2]. Using the 5620 as the real-time controller, we are able to place a synthesizer under the control of a host computer (in this case, a VAX 750), without a large amount of systems programming.

The MIDI device driver (*midiblt*) is down-loaded into a *mux* window, where it takes over interrupt handling on the parallel I/O port. Three types of interrupts are handled: UART transmitter and receiver ready, and a 5 msec clock. MIDI data are organized into three queues. Incoming messages are time-stamped and placed on the *receiver* queue, where they are available to other software. At clock interrupt, the *scheduler* queue is examined for messages with time less than or equal to the current time; such messages are moved from the scheduler queue to the *transmitter* queue, from which they are sent to the hardware as quickly as possible.

There is a sharp division between routines which control the UART directly (and handle single characters) and those which manipulate MIDI messages. For example, the MIDI transmitter is a finite-state machine which understands things like elided status bytes; it is called by a lower-level routine and produces the next byte to be sent by the UART.

Routines which empty the receiver queue and fill the scheduler queue are available from outside the driver. *Midiblt* places their entry points in a name table maintained by *mux*; since there is no memory mapping in the 5620, they are immediately available to other programs down-loaded in the terminal. This code sharing mechanism is used to implement a virtual-keyboard program, *jx7*, which provides the functionality of a one-fingered mouse-pianist, with slide controls for volume, vibrato, pitch bend, etc.

3. Communication with the host

Host communication takes place via the 8th Edition *stream* mechanism, described in detail in [3]. Briefly, each window is associated with a control stream /dev/pt/pt*nn*, managed by *mux* on the host; when a program down-loaded in the terminal does a read or write, *mux* performs the appropriate operation on the pt associated with the window. The program at the other end of the stream can be the shell (this is how multiple virtual terminals are implemented), or a special-purpose program (as in the case of a text editor). *Midiblt* falls into the second category. It makes a *symbolic link* (another 8th Edition feature) to its pt under the name .MIDI in the user's home directory, in effect creating a character-special file for the MIDI driver. The terminal side of *midiblt* reads characters as they appear from the host, assembles them into MIDI messages, and places the messages on the scheduler queue. (The host side of *midiblt* does not look at this data; error correction and flow control are performed by *mux*).

We have followed a time-honored Unix tradition by formatting the MIDI file in ASCII. Such a file consists of a series of lines, one per MIDI message. Each line is a sequence of blank-separated decimal numbers, viz.: event time (msec), status byte, and data. (Backslash-newline can be used to break long system-exclusive messages.) In this form, the entire panoply of Unix text-processing tools can be brought to bear in rough-and-ready fashion. On the other hand, this format is not well suited for direct transmission to the 5620, since bandwidth is at a premium. The program *midi*, which compresses the data, is used; it replaces (and has similar semantics to) "cat >.MIDI".

In addition, *midi* attaches itself to the *process group* associated with the .MIDI pt, which allows the *midiblt* window to send signals to the *midi* process; this is necessary so that, e.g., if the user does a reset from the *midiblt* menu, there is proper coordination between the host and the driver. The *midi / midiblt* pattern is repeated in the programs *score* and *scoreblt*, which produce a pitch vs. time graph. *Scoreblt* manages a terminal window and draws the display. Through the code-sharing mechanism mentioned earlier, it has access to *thinkblt* which drives a Hewlitt-Packard ThinkJet dot-matrix printer.

4. Synthesizer control

The system described was first used to run a Yamaha DX7 and, later, a Yamaha TX816. Both produce sound via FM synthesis [4], a voice being described by around 100 (digital) parameters, all settable by system-exclusive MIDI messages. The DX7 has internal memory for 32 pre-loaded voices which can be selected by number (via the MIDI *program change* message). It is a keyboard instrument with additional controls for editing voice parameters (the current voice, whether internal or downloaded, is always copied into an editing buffer). Although the DX7 is limited to playing in one voice at a time, up to 16 simultaneous notes can be produced. The TX816 is a rack-mounted unit consisting of eight TF1's (essentially a DX7 without keyboard). Each TF1 can be set to a different MIDI channel, so that orchestral and multi-track effects are possible.

A number of C programs are used for synthesizer configuration. *Txchan* assigns channel numbers to the TF1's. *Mecho* is used to send "constant" data (such as to select an internal voice, or alter single parameters of a voice). *Dxvoice* downloads voice data from a library file on the host. Here is an example of a shell script which sets up the TX816 with five instruments, one of which (the harpsichord) uses two TF1's in parallel. The violin voice needs to be downloaded; the rest are already stored in the proper TF1's (they came from the factory this way). Percent signs delimit comments to *mecho*:

```
VLIB=/usr/tom/midi/dx/voices
txchan 1 1 3 4 5 6 7 8
mecho   init \
        prog    -c1 28          % harpsichord chan 1   % \
                -c4 3           % reeds chan 4         % \
                -c6 24          % flute chan 6         % \
                -c8 3           % bass pipes chan 8    % \
        dx      -c8 p144 24     % move up an octave    % \
        parm    -c4 p4 127      % foot control reeds   % \
                -c6 p2 127      % breath control flute % \
                -c8 p4 127      % foot control pipes   %
dxvoice -c3 -v2 $VLIB/tx816.8   # solo violin chan 3
```

This scheme is complete in that it allows access to any MIDI or DX parameter, but it is not altogether satisfactory. The user must know, for example, that voice 3 is "reeds" in the fourth unit, but "bass pipes" in the eighth. Simply assigning names to numbered parameters does not reduce the complexity, however. Ideally, one would like an interactive "orchestration editor" supported by a large database.

5. Musical examples in C and the shell

The first piece to be played on our system was written by Cynthia P. Killian using a combination of C and the Bourne shell. The composer took raw material produced by the "munching squares" algorithm and manipulated it by splicing and dubbing techniques. The heart of the algorithm is listed here:

```
main()
{
        int c, x, y, z, magic = 13, size = 64;
        int mask = 2*size-1;
        for (c = z = 0; c <= mask; c++, z += magic)
                for (newfile(), y=0; y < size; y++)
                        if ((x = (y ^ z) & mask) < size)
                                play((x+y), (x-y));
}
```

X and *y* trace out short diagonal segments, which are rotated by 45 degrees and passed to *play*. The resulting vertical coordinate is mapped onto a tone row, and the length of the segment (as determined by a sequence of points at the same height) determines the length of time the tone is held. *Newfile* breaks the output into separate small files (smunch.??) for convenience. The latter are then processed by shell scripts, for example, this one:

```
(divider <smunch.06 3 4; divider <smunch.07 1 2
 divider <smunch.07 1 2 |
  invert 0 1120·0 | retro) | ttrans 0 280 >tmp$$
(cat tmp$$
 divider <smunch.08 7 8) | ttrans `endtime <tmp$$` -1260
rm tmp$$
```

The unfamiliar commands in the script are either shell scripts or trivial C programs. *Sed* and *sort —n* form the basis for cutting and pasting. Operators which do a lot of arithmetic (e.g. *retro*, which time-reverses its input) are written in C. The shell syntax for operator precedence and the semantics of the Unix filter are extremely well matched to this application.

6. The M language

The need for a closer tie with standard musical notation led to the development of the M language. We will try to give a feel for the language with a short example, the beginning of a Bach invention:

```
/* Inventio 4 (BWV 775) */
8 = 120        /* tempo: 120 eighth notes/min  */
b@             /* key signature: F             */
3 / 8          /* time signature               */
{
soprano : 1     ¦ F100 16: d3 e f g a b ¦ c# b a g f e ¦
bass : 1        ¦ F100 4.: r            ¦ r            ¦

soprano ¦  8: f    a   d4  ¦ g3   c4# e   ¦ 16: d e f g a b ¦
bass    ¦ 16: d2 e f g a b ¦ c# b a g f e ¦  8: f   a   d3  ¦
...
}
```

An M file consists of "front matter" followed by text enclosed in matching curly braces. The comment convention is the same as in C. The music is divided into *lines* formed by a *voice name*, an optional colon and MIDI channel number, and one or more *measures*. Voice names are arbitrary alphanumeric strings. Measures are delimited by barlines and contain *notes*, *rests*, and *modifiers*. Notes specify *pitch class* and, optionally, *octave number* and *time value*. Pitch class is indicated in standard letter notation, with accidentals #, @ (flat), and = (natural). The octave is given by a number appearing after the letter name, e.g., c3 is middle c, and c3#, c#3 are a half-tone above. The octave number changes between b and c, so a half-tone below middle c is b2. A missing octave number defaults to that of the previous note in the same voice. Time value is given by a number preceding the letter name (1 = whole note, 2 = half note, 4 = quarter note, 4. = dotted quarter note, etc.). A default value may be set, as in the example above, using a time value followed by a colon. Rests have time value only and are indicated by the letter "r."

A default time value is an instance of a modifier; another modifier used in the example is "F100" which specifies a "key force" (volume) of 100 units.

Not shown in the example are notations for ties and more complicated rhythms. Ties map particularly easily onto MIDI events. Observe that a note usually generates two MIDI events, note-on and note-off. A note with a tie going out (e.g., c—), simply loses its note-off event, and one with a tie coming in (e.g., _c) loses its note-on event. Time values that are inconvenient to generate otherwise can be specified with numerator and denominator, e.g., 9/17.

Also not shown in the example are notations used for grouping. Parentheses group a *sequence* which may take a modifier, e.g., "8(c e g)" is a sequence of eighth notes. Square brackets group a *chord* similarly, e.g., all of the notes in "[c e g]" are sounded simultaneously. Groupings may be nested, e.g., "[(c d c) (e g e) (g b g)]" has three sequences running in parallel to make a I-V-I progression.

M is based on *lex* and *yacc*, and its continuing development depends heavily on them. M is intended to be used by musicians who are not computer scientists, and who can't always pass on the merits of a feature without testing it. Hence the ability to experiment is most important. *Lex* in particular has been justly criticized on performance grounds, given that lexical analyzers are fairly easy to write by hand. But this is the case only for a static language, and M is still changing rapidly, particularly in the area of dynamics control.

M produces a MIDI file on its standard output, so that "m bach ¦ midi" is an example of a common invocation. It also has options to restrict the output to certain voices or a range of measures, as a debugging aid.

7. The M keyboard interface

It is possible to generate the notes of an M program by playing at the DX7 keyboard. First the note-on and note-off events are collected by *midiblt* and written into a file on the host. This file is then converted into a list of note names by *unmidi*, with the convention that the highest note on the keyboard (c6) generates a newline. This gives the user some control over the format. *Unmidi* also has options to direct it towards desired enharmonic spellings. Next the output from *unmidi* is fed to an *awk* script which adds voice names. Now only the rhythm and internal barlines are missing. The latter can be omitted if desired (sum-rule diagnostics will not be available). The rhythm can be added in the format already described, but an alternative method (suggested by an irate user) has proven much faster. A list of rhythmic values (including rests) is enclosed in angle-brackets and placed between the voice name and the notes, as in this fragment from a Monteverdi madrigal:

```
canto    < 1 5(1r) 4. 8 2 4 8 8 2-8 >
         | c4 e3 e f f# f# f# g |
quinto   < 1 4(1r) 2. 4 2. 8 8 2-8 8 8 8 >
         | g3 b b c4 c# c# d d d d |
alto     < 1 3(1r) 2r 8r 3(8) 4-16 3(16) 4. 8 4 4 2r 1r >
         | e3 a a g g f= f f f e e e |
tenore1 < 1 2(1r) 2. 4 2 4 8 8 2 4(8) 2 2 1r >
         | c3 e2 e f f# f# f# g g# g# g# g# a a |
tenore2 < 2 2. 4 2. 8 8 2-8 3(8) 3(2) 4(8) 4-16 3(16) 4 4 1.r >
         | g=2 e e f= f# f g g g# g a a r r e3 e d d d d d c c |
basso    < 1 2. 4 2 4 8 8 2 8 8 8 8 2 2 1r 1r 2. 4 >
         | c2 a1 a b@ b= b b c2 c# c# c# c# d d b1 b |

violin1 < 1 7(1r) >
         | e4 |
violin2 < 1 7(1r) >
         | c4 |

figbas1 < 1 7(1r) >
         | c4 |
figbas2 < 1 2 2. 4 2. 4 2-2. 4 2 2 2. 4 1 >
         | g3 e f= f# g g# a b b b c4 c# d |
figbas3 < 1 1 1 1 7(2) 2-4 >
         | e3 c d e f= f# d f= e f f# g= |
figbas4 < 1 2. 4 8(2) 2. 4 2 2 >
         | c2 a1 a b@ b= c2 c# d d g g# a a b1 b |
```

In this form, the rhythms can be typed in quickly from an existing score. Parentheses indicate multiples, so, e.g. 7(1r) means seven whole rests. In addition, — can be used as a tie to add time values together. The list of time values is put in one-to-one lexical correspondence with the notes and the result is compiled as before.

8. Future development

The most glaring deficiency in the system we have described is the absence of a true score editor. This is expected to be the next major development effort. The editor will read and write M files, so that an ordinary text editor can still be used if suited to the task at hand.

Twelve-tone pieces such as the one in section 5, and serialized pieces particularly, would be easier and faster to develop with the aid of specialized tools. These could range anywhere from a library of C routines to a complete language implementation. We expect to begin on a modest scale with the former and enlist the efforts of interested composers.

Voice creation is another important area. A voice editor should provide graphical control over envelopes and other parameters, and work with the virtual-keyboard program to give immediate audio feedback.

It is clear that we have barely scratched the surface of an immensely challenging class of problems. The Unix system, originally crafted as a home for programmers, has proven remarkably robust, flexible, and downright hospitable as a base for an exotic application.

9. References

[1] *MIDI Specification*, document no. MIDI-1.0, August 1983. International MIDI User's Group, P.O. Box 593, Los Altos, CA 94022, USA.

[2] R. Pike, "The Blit: A Multiplexed Graphics Terminal," AT&T Bell Laboratories Technical Journal, vol. 63, no. 8, part 2 (Oct. 1984), pp. 1607-1632.

[3] D. M. Ritchie, "A Stream Input-Output System," AT&T Bell Laboratories Technical Journal, vol. 63, no. 8, part 2 (Oct. 1984), pp. 1897-1910.

[4] J. M. Chowning, "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation," Journal of the Audio Engineering Society, vol. 21, no. 7 (1973).

IBM Advanced Interactive Executive (AIX) Operating System

S Mecenate

1. IBM AIX STRUCTURE

The IBM 6150 micro computer is a new synthesis of computer concepts. It combines:

- a very fast reduced instruction set 32-bit processor for efficient execution of programs compiled from a higher-level language.
- a resource manager to provide virtual machine, storage, and I/O functions and to ensure data integrity and processing continuity,
- a multitasking, multiuser operating system that can be tailored to make the 6150 suitable for a variety of user requirements,
- a coprocessor feature that allows users to run programs written for the IBM PC without interfering with the normal operation of the 6150,
- a wide variety of displays, printers, communications adapters, and processing features,
- in a box that fits on or under a desk.

As the base for the 6150 Advanced Interactive Executive (AIX[1]) operating system, IBM chose AT&T's UNIX[2] System V, because it provides considerable functional power to the individual user, provides multi-user capabilities where needed, is open-ended, and has a large user and application base.

In choosing UNIX, however, we accepted the need to make significant extensions and enhancements to meet the needs of our expected customers and target applications.

Some of the major enhancements made were:

- a usability package to provide easier access to the capabilities of the UNIX command language and to simplify the implementation of full-screen dialogs.
- multiple, full-screen virtual terminal support to permit a single user to run several interactive applications concurrently, time-sharing the console display.
- enhanced console support including extended ANSI 3.64 controls, colour support, sound support, and mouse support.
- an indexed data management access method that is integrated into the base UNIX file system structure (this allows UNIX system utility functions such as "cp" to transparently operate on composite data management objects consisting of an index file and a data file)
- extensions to exploit use of the powerful virtual storage support in particular, mapped file support which allows an application to "map" a file into a 256 megabyte virtual address space, and access it with loads and stores, versus reads and writes, (a derivative is used by the system to provide mapped text segment

[1] "AIX" is a trademark of International Business Machines Corporation

[2] Trademark of AT&T Bell Laboratories.

support, allowing paging "in place").

- enhanced signals to allow flexible exception-condition handling
- a variety of floating point support functions
- simplified installation and configuration processes.

In light of our requirements for application diversity, operating system stability, and exploitation of the 6150's advanced hardware features, we felt that the best approach was to provide enhancements below, within, and above the kernel. This led to the software structure shown in Figure 1. The Virtual Resource Manager (VRM) controls the real hardware and provides a stable, high-level machine interface to the advanced hardware features and devices. The kernel received corresponding enhancements to use the services of the VRM and to provide essential additional facilities. The application development extensions above the kernel were integrated into the existing operating system structure.

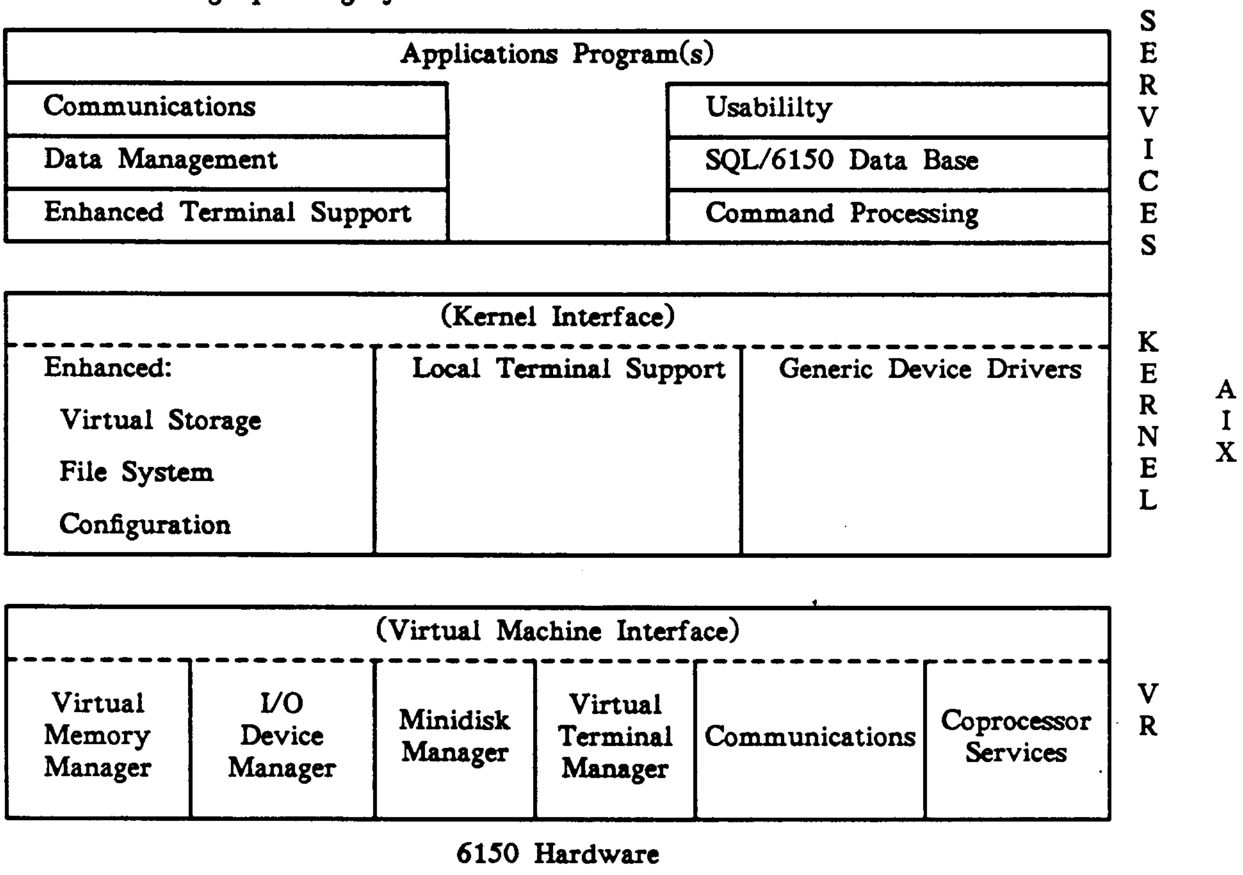


Figure 1. Overall Structure of AIX Operating System

Although the VRM and the AIX kernel have been "tuned" for each other, we have not precluded the ability to run other operating systems in the VRM virtual machines. Similarly, the techniques we used to virtualize the existing types of devices would work for new device types as well. Both the VRM and the kernel are deliberately open-ended to allow the straight forward addition of new functions and device support.

The existing structure of the AIX kernel was not well suited to exploit the advanced features of the 6150 hardware. Rather than making major changes to the architecture of the kernel. VRM is built to provide a more comprehensive real-time execution environment. This environment includes multiple preemptable processes, process creation and priority control, dynamic run-time binding of code, direct control of virtual memory, millisecond-level timer control, multiple preemptable interrupt levels, and an efficient interprocess communication mechanism for main and interrupt-level processes. The VRM software uses these features to control the ROMP processor, Memory Management Unit (MMU) and I/O hardware, and provide the kernel with interfaces to these functions.

The key to the ability of AIX to support multiple simultaneous interactive applications is the virtual terminal. A virtual terminal is a virtual counterpart of the real 6150 display(s), keyboard, and mouse. Each application initially gets a single virtual terminal to work with. The application can request creation of additional virtual terminals at will. The virtual terminals time share the use of the real displays and input devices. A virtual terminal can function as either a simulated ASCII terminal or a high-function terminal equivalent in power to the real hardware.

The ROMP/MMU virtual memory architecture, in combination with the VRM, gives the 6150 a demand-paged virtual memory of one terabyte, consisting of 4096 256-megabyte segments. The VRM performs page fault handling and manages the allocation of real memory, paging space, and virtual storage segments. It provides the AIX kernel with interfaces to control these functions and to respond to a page fault by dispatching another process. The VRM can also map memory pages within a given segment onto disk file blocks, creating a "single-level store" that makes DASD and memory equivalent.

The VRM provides the operating system with an extensive, queued or synchronous interface to the I/O devices, insulating the kernel from the details of specific devices and the management of share devices. The correct device handler is selected on the basis of the currently-installed hardware or the configuration files and is dynamically bound into the VRM. The devices that the application sees are generic devices such as generalized fixed-disk drives ("mini-disks") or RS232C ports. In those cases where the generic devices are not appropriate, or where the real time capabilities of the VRM environment are needed by the application, the user or a third-party programmer can write C or assembler language code to implement the necessary function, and can dynamically add that code to the VRM while the VRM is running (i.e without re-IPL).

Problem determination in system or user-added code is supported by VRM serviceability facilities that include trace capabilities, dumps, and a debugger.

The VRM supports the PC AT coprocessor option as though it were another, rather specialized, virtual machine. The coprocessor runs concurrently with the execution of programs in the ROMP, but it only has access to the keyboard, locator, and display when the coprocessor virtual terminal in the "active" virtual terminal, that is, when it has control of the display. The input from the keyboard and locator are presented to the coprocessor as though they had been produced by the corresponding PC at devices. If no display has been dedicated to the coprocessor, the display interface emulates a PC display on the system display. The VRM manages the shared system resources to ensure that the ROMP and coprocessor operate cooperatively.

The VRM resides on a minidisk of its own in a standard AIX file system. Installation and space management on that minidisk are performed with standard AIX utilities.

To be able to support the full range of modern applications, AIX needed several functional extensions. One of the most critical was the need for an indexed access method. We added a B-tree based data management program that permits either record-level of field-level access. Although it is packaged separately from the operating system, data management becomes an integral part of the file system when it is installed. Similarly, we added a data base program supporting the Structured Query Language (SQL) to provide both users and application programmers with relational data base facilities.

2. AIX MODIFICATIONS AND EXTENSIONS

The structure of AIX reflects our response to several key objectives for the 6150.

- a primary use of the 6150 was expected to be as a personal workstation.
- we had to ensure that the performance potential of the 6150 was achieved.
- the system has to be tuned to operate effectively in a virtual memory environment.
- the kernel had to be made robust enough to be the centre of a production operating system.

The following sections describe the various changes and additions that were made to meet our objectives.

2.1 Appropriate Interfaces For A Personal Workstation Environment

2.1.1 Auto Logon

The auto logon facility permits a user to be automatically logged on at the systems console. This facility is intended for a single-user system or for those users who are the only ones to logon at the systems console.

Auto logon is performed when the file /etc/autolog contains the name of a valid login name as its first or only entry.

2:1.2 Multiple Concurrent Groups

The multiple concurrent groups facility allows a user to access files that are owned by any of the groups in which the user has membership. The "primary" group is specified in the /etc/passwd file. Any additional groups are specified in the /etc/group file. The setgroup system call is used to specify to the kernel all of the groups of which the user is a member. If the user would not normally be granted access to a file on the basis of the standard permission checks, the user's group access permissions are checked. If the user is a member of the group that owns the file, access to the file is granted.

The system makes extensive use of this facility in controlling and permitting access to certain privileged system files.

2.1.3 Reduce Superuser Dependency

The AIX system is configured to allow a user to perform many of the superuser functions without having to log on to the system as superuser or to issue the su command.

This scheme is based on the use of the AIX file permissions, making extensive use of group permissions, multiple concurrent groups, and set user ID.

Each of the files (commands, data, etc) is assigned to a particular group, and users are assigned to corresponding groups (staff group or system group) depending on the authority to be given to the particular user.

2.1.4 Removable Media

The removable mount facility of AIX is intended primarily to be used with diskettes which contain mountable file systems. With this facility mountable diskettes may be inserted in and removed from the diskette drive without doing an explicit mount or umount command.

2.2 Interactive Workstation

The Interactive Workstation (IWS) program allows the user to easily connect, via the asynchronous ports, to another computer system, from either the AIX system console ports or an attached terminal. The connection can be initiated via a command interface or a menu-driven interface. The following functions are provided:

- The system console keyboard appear as either a 6150 or an asynchronous terminal to the remote system
- two protocols to transfer files to or from the remote system
- capture received data in a system file as well as display the data on the user's screen
- a phone directory function which is maintainable by the user
- a menu by which the user can alter the local terminal characteristics
- a menu from which the user can alter the data transmission characteristics
- utilize any of the supported asynchronous communication adapters
- connect to another 6150 and invoke IWS on that system to connect to a third system
- allow two users on a given system to concurrently use IWS
- invoke IWS or XMODEM from another 6150 or terminal by dialing into an AIX logger.

The menu-driven interface to IWS consists of several menus. The main menu is first displayed when IWS is invoked. This menu allows the user to request:

- a connection to another system
- a phone directory menu
- help information
- the "modify local terminal variables" menu
- an operating system command
- quit the IWS program

The connection menu allows the user to:

- send a file
- receive a file
- send a break sequence
- terminate the connection

3. Efficient Operation In A Virtual Memory Environment

3.1 The Virtual Machine Environment of AIX

The AIX operating system kernel executes in a virtual machine maintained by the Virtual Resource Manager (VRM). The VRM provides virtual machines with paged virtual memory, up to 2^{40} or one terabyte. The effective addresses generated by instructions are 32 bits long, with the high-order 4 bits selecting a segment register and the low-order 28 bits providing a displacement within the segment. The segment registers contain a 12-bit segment ID. The 12-bit ID plus the low order 28 bits of the effective address yield the 40-bit virtual address. A virtual machine may have many segments defined. To access one of these segments, the virtual machine loads a segment identifier into one of the 16 segment registers. Segments are private to a virtual machine unless the virtual machine that creates the segment explicitly gives other virtual machines access to that segment.

Pages consist of 2048 bytes. A segment can contain from 1 to 131,072 pages. Protection is available at the page level. Pages are brought into active storage on a demand basis via page faults.

3.2 Virtual Memory Program Management Extensions

The AIX kernel has been enhanced to use the VRM virtual memory services. Three AIX program management extensions take advantage of the advanced virtual memory support.

3.2.1 Segment Register Model

At any given time, the IDs of up to 16 segments may be loaded into the segment registers. Each of the 16 segments may be up to 256 megabytes. Each page in a segment is individually protected for kernel access and user access. The AIX kernel occupies segment register 1. Each user process is allocated three segments. Segment register 1 is used for the user text segment. The user data segment occupies segment register 2, and has read-write access. Segment register 3 is used for the user stack. Segment registers 4 thru 13 are used for shared-memory segments and for mapped data files. Segment register 14 is used by the VRM to perform DMA operations. Segment register 15 is used to address the I/O bus directly.

3.2.2 Demand Paging of Both Users and AIX Kernel

Both the users and the kernel execute in demand-paged virtual memory. When a user-process results in a page fault, the VRM notifies the kernel, so that another process can be dispatched. This page fault notifications results in improved overall systems performance. Page faults which occur for a kernel process are handed synchronously, with no preemption of the kernel process.

3.2.3 Process Fork Enhancements

The AIX "fork" system call creates a new process. The new process (child process) is an exact copy of the calling (parent) process's address space. The address space consists of text, data, and stack segments. Typically, when executing a new comand, the "fork" system call is followed by an "exec" system call to load and execute the new command in the new copy of the address space. This results in replacing the forked address space with the address space of the new command, thus undoing much of the work of the fork.

The VRM "copy segment" SVC creates a new segment, but delays the actual copying of the data until one of the sharing processes actually references the data. Therefore, most of the data will not have to be copied when an "exec" system call follows, thus saving the time and memory required for the copy. the AIX "fork" system call uses this VRM copy-segment facility to create the segments of a new process. This enhancement of "fork" reduces wasted effort.

3.3 AIX and Mapped Data

3.3.1 Mapped Page Ranges

Simple paging systems usually suffer from conflicts between file I/O and paging I/O. For example, a file device driver may read disk data into a memory buffer, then the paging system might write that buffered data out to disk.

Potential duplication of effort also exists with program loading. Having the VRM page the program directly from the program library saves having to explicitly load programs and also eliminates space wasted by copying the program out to a page area of the disk.

Carried to the extreme, only the paging system would need to be able to do physical I/O. The AIX file manager could tell the VRM the mapping between data on the disk and virtual memory pages, and the paging system could then perform all the physical disk. I/O.

The close interaction between the AIX kernel and the VRM offers several distinct advantages

- reduction in secondary paging space
- improvement of performance
- simplifications of the data addressing model.

The VRM supports a means by which AIX can map the disk block of a file to a virtual memory segment and have physical I/O performed by the memory management component of the VRM. This mechanism is known as "mapped page ranges".

3.3.2 Mapped Executables

The AIX kernel implements mapped page range support in the form of mapped executables. When a program is loaded, the AIX kernel maps the program's disk blocks to distinct memory text and data segments. Only the program file header is "read" by the kernel. All remaining disk I/O is demand-paged as the program is executed. This results in a significant performance increase for large programs.

3.3.3 Mapped Data Files

AIX mapped file support consists of a system call interface to the data file map page range facilities. The "shmat" system call, with the SHM MAP flag specified, is used to map the data file associated with the specific open file descriptor to the address space of the calling process. The data file to be mapped must be a regular file residing on a fixed-disk device.

When a file is mapped onto a segment, the file may be referenced directly by accessing the segment via Load and Store instructions. The virtual memory paging system automatically takes care of the physical I/O.

A significant amount of system overhead is eliminated by mapping a data file and accessing it directly via load and store operations, rather than conventional access via "read" and "write" system calls.

4. Building A "Production" Operating System

A number of enhancements were needed to make AIX suitable for the wide variety of customer environments and applications.

4.1 I/O Management

We restructured the I/O management area of the kernel to make effective use of the VRM's I/O facilities. Instead of a specialized device driver for each distinct device, we created a family of generic device drivers that are capable of supporting a number of unique devices of a given class. For example, a single "async" device driver handles async, RS232C, and RD422 interfaces. Truly device-specific considerations are left to the VRM device drivers, which can be added or replaced dynamically without bringing down the system.

4.2 Multiplexing

We added a facility to allow dynamic extensions to a file system. If the multiplex bit in the special file inode is on, the last qualifier of the file name is passed to the character device driver. The driver looks for the file outside of the nominal file system. This facility is used to deal with virtual terminals and communications sessions as files.

4.3 File Systems

The AIX file system takes advantage of the virtual device interface provided by the VRM. To improve performance, we increased the block size of the file systems and the buffer cache to 2048 bytes. To permit AIX to accommodate an indexed data management feature and a data base manager, we added the ability to synchronize the buffer cache with the fixed disk on a file rather than a file system basis, added locking facilities, and incorporated facilities to recover space in sparse files.

4.3.1 Use of Minidisks

The VRM provides the ability to divide a given fixed disk into a number of minidisks. This permits the seperation of file systems for different purposes onto different virtual devices.

AIX uses 512-byte blocks for diskette file systems and 2048-byte blocks for disk file systems.

The space on each minidisk that contains a file system is divided into a number of 2K-byte blocks. A corresponding cache of 2K-byte buffers is used to reduce re-reading of blocks.

4.3.2 Buffer Cache Synchronisation

Cache buffers are normally only written to permanent storage before the buffer is used again or with the "sync" system call. AIX has, in addition, the "fsync" system call that works on an open-file basis to force the modified data in the cache buffer to permanent storage and does not return until all of the buffers have been successfully written. This gives the user more control over the data on the disk and permits an application such as data management services to force writing on only those buffers that really need to be flushed.

4.3.3 Dynamic Space Management

AIX has two system calls to recover space within once-sparce files. The calls are "fclear" and "ftruncate".

- fclear - zeros a number of bytes starting at the current file position. The seek pointer is advanced by the number of bytes. This function is different from the write operation in that it returns full blocks of binary zones to the file by constructing holes and returning the recovered blocks to the free list of the file system.

- ftruncate - removes the data beyond the byte count in a file. The blocks that are freed are returned to the free list of the file system.

4.3.4 File/Record-Level Locking

AIX file and record level locking extensions allow an individual file to be locked in either an advisory or enforced form.

Records may be of any length ranging from one to the maximum of the file size. Locks may be applied beyond the current end-of-file or over an area that has not been written (sparse file regions).

4.4 Process/Program Management

4.4.1 Signals Enhancements

In addition to the standard set of System V signals, AIX provides an enhanced signal facility. The following are brief descriptions of the system calls that make up the enhanced signal facility:

SIGBLOCK adds specific signals to the list of signals currently being blocked from delivery.

SIGSETMASK sets the signal mask (the set of signals to be blocked from delivery) to a specified value.

SIGPAUSE sets the signal mask to a new value, pauses until a signal not blocked by the mask is received, and restores the signal mask to its original value.

SIGSTACK allows users to define an alternate stack to be used for signal handling or get the state of the current signal stack.

SIGVEC allows users to specify how a specific signal is to be handled.

EXECVE starts a new program in the current process, resets all signals that are being caught by the original program to terminate the new program, resets the signal stack state, and leaves the signal mask untouched.

4.4.2 Buffer Bypass Variations

"Buffer bypass" is a form of disk I/O which, like raw I/O and mapped files, bypasses the kernel's buffer cache, transferring data directly between the VRM disk device driver and AIX user processes. this offers direct and indirect performance gains when it is unlikely that the data will soon be reaccessed. The direct gain is the lack of a memory-to-memory copy of the data. The (more substantial) indirect gain is the generally improved cache hit ratio which results from not replacing useful cache blacks with data that is unlikely to be reused.

4.4.3 IPC Queue Extensions

System V Interprocess (IPC) message services have been extended to give more information when receiving ipc messages. The new function call is "msgxrcv".

The msgxrcv function returns an extended message structure that contains the time the message was sent, the effective user ID and group ID of the sender, the node ID of the sender or zero if the sender was on the local mode, and the process ID of the sender.

Applications and servers can use the additional information found in the extended ipc message structure to check permissions and send time.

4.5 Terminal support

AIX terminal support is tailored to work in the VRM environment, where terminals are virtual constructs rather than real devices. It permits applications to use multiple virtual terminals and to access their virtual terminals in either extended ASCII mode or in "monitored" mode.

4.5.1 Console Support

In order to take advantage of the unique functions provided by the virtual terminal manager subsystem of the VRM, a console device driver was created and modeled on the RS232 terminal device driver (tty). This new device driver is referred to as the HFT device driver. It provides support for a console consisting of a keyboard, mouse or tablet, speaker, and up to four displays.

4.5.2 Multiple Virtual Terminals

Some device semantics were established to allow programs to create new virtual terminals and access existing ones. If a program wishes to creat a new virtual terminal, the open system call is issued on the device /dev/hft. If an existing virtual terminal is desired, the program opens the device /dev/hft/n, where n is the character representation of a decimal number.

If a program needs to know about and control activity on all the virtual terminals associated with the console, it opens the device /dev/hft/mgr. This gives the program access to the screen manager component of the VTM subsystem. The program may now query the state of all the virtual terminals, activate any terminal, or hide any terminal by issuing an ioctl.

4.5.3 Extended ASCII Mode

The default mode for a virtual terminal simulates an enhanced version of the standard ASCII terminal. It permits programs built for that infterface to run with minimal change. It also permits new versions of such programs to access the sound and mouse functions.

4.5.4 Monitored Mode

In order for a program to operate a bit-mapped display in bit mode, the program deals directly with the hardware display adapter by storing to the memory-mapped I/O bus.

Even though operating a terminal in monitored mode is complex, the speed of direct hardware access is attained, and the protected environment of a multiuser system is preserved.

4.6 Printer Support

4.6.1 Device Driver

The printer device driver provides the interface to the VRM from the kernel environment. Up to eight concurrent printers are supported. Enhancements have been made to provide better error recovery procedures. Errors, as they are discovered, are returned to the application environment only if the application requests that they be returned. A new set of ioctls has been defined to allow printer control from the application.

Printer performance has often been a problem when high speed printers were used. The device driver noe supports both synchronous and synchronous write system calls. Each of these functions is performed for both serial and parallel printers.

By adding serial printer support to the printer driver, the full performance and error recovery enhancements can be utilized.

4.6.2 Replaceable/Addable Backends

The print command allows user access to the queuing environment. Multiple queues per printer allow the same printer to be used for different job types. Mutiple printers per queue can keep the output flowing in case one printer is unusable. The user should not have to know the details of how each printer works. By providing a more general printer-support structure, we made it easier for the user to install and use one of the new IBM printers without knowing the details of how it works. Configuration options allow printers to be set up for different types of jobs. Thus, existing applications will work on the new printers without changing the application.

4.6.3 Extended Character Set

The use of the 7-bit ASCII code definition in 8-bit-byte machines has created some problems. AIX displays and printer support for 8-bit codes was implemented to help meld PC applications into the world of AIX. The 8-bit support is compatable with 7-bit ASCII applications and provides an additional degree of commonality with a large number of PC applications and files.

The AIX support provides a canonical mapping of the most widely used IBM code pages required by scientific and international applications. The display and printer subsystems provide controls for accessing these code points. Data stream controls provide switching to one of three code pages. These code pages are designated: P0, P1 and P2.

The base code page, P0, is based on the IBM PC display font with the exception that the first 32 code postions contain controls instead of graphic characters. In order to access graphics on code pages P1 or P2, application programs need to imbed switching controls for the printer or dispay in the output data stream.

The extended graphic characters defined in P0, P1 and P2 fulfill the major support requirements for the US, Europe, Teletext, and scientific symbols.

4.7 Floating Point Support

The 6150 system provides enhanced services for floating point arithmetic. These services are designed to support the Institute of Electrical and Electronic Engineers' (IEEE) new standard for Binary Floating Point Arithmetic (754). The floating point package is utilized by the C, FORTRAN, and Pascal compilers for all floating point operations. Floating point operations can be further enhanced with the addition of the hardware floating point accelerator feature.

The floating point routines provide an environment of six floating point registers, with a status register that controls exception and rounding modes. The floating point registers may contain either a single-precision or a double-precision floating point number.

The Floating Point Accelerator has 32 sets of floating point registers available for user processes. When there is no Accelerator, floating point subroutines emulate the floating point registers in a reserved area on the user's stack.

5. Reliability/Availability/Serviceability (RAS)

The IBM 6150 system RAS support is designed to provide a coherent and consistent set of error detection and correction schemes. Wherever possible, functions and components are self-diagnosing and correcting; that is:

- error messages with clear unambiguous meaning are generated
- formatted error logs are automatically generated
- dump facilities are provided
- error analysis routines support software and hardware problems determination

A primary objective of the 6150 system RAS support is to provide problem determination and correction. As such, the system must be reliable in all respects, but in the event that there is a failure, the system must be easily and quickly diagnosed and recovered.

The following are problem determination facilities in the 6150 system.

5.1 Trace

The trace function is intended to provide a tool for general system/application debug and system performance analysis. Trace monitors the occurrence of selected events in the system. Important data specific to each of these events is recorded on disk. When the user needs to view this data, a trace report program formats the trace data in an intelligible form. The trace function may be started either by the user or by an application.

Trace can operate at all levels of the system: below the VMI, in the kernel, and at the application level.

5.2 Dump

The IBM 6150 system provides a system level dump capability to enhance the user's ability to do problem determination and resolution. In the IBM 6150 a "DUMP" environment may be characterized in several ways:

- the VRM or virtual machine ceases execution
- the VRM or virtual machine abends

These failures may occur in an application, the base operating system, or the VRM.

When a failure occurs, the user may choose to initiate a dump. The user presses a dump key space sequence: CTL-ALT-NUMPAD8 for a VRM dump. The target for a virtual machine dump is the dump minidisk and the data placed on that dump minidisk is defined by UNIX System V. The target for a VRM dump is a high-capacity diskette.

The VRM dump program is permanently resident in memory. It has its own diskette device driver. It is self-contained and does not depend on any VRM resources.

5.3 Error Log

The error log function provides a tool for problem determination of hardware and some software errors. Data specific to a problem or potential problem and certain informational data is recorded on disk. When the user needs to view this data, an error report program formats the error log data in an intelligible form.

For each hardware entry in the error report, an analysis of the error is appended. This specifies the probable cause, the error, what hardware pieces to suspect as bad, a list of activities the user could perform for further isolation, and a service request number. This analysis is based solely on that error entry. The error log function can be started by superuser, but is generally started by /etc/rc.

Error logging can operate at all levels of the system: below the VMI, in AIX, and at the application level.

5.4 Update

Updates for software products on the 6150 are packaged together on the same diskette. A new "update" command provides a menu interface to applying these updates. When an update diskette is received, the user can "apply", on a trial basis, the updates for one or more of the software products that are already installed on the system. The user can then test the updated programs to ensure that they still function correctly in that environment. If the updates have caused a regression, the user can run the update command to "reject" (back out) the update. Otherwise, the user issues the update command to "commit" the update as the new base level of the program.

6. Conclusion

We believe that we have successfully made AIX into an operating system that can be used without detailed knowledge of its internal structure. It takes advantage of the functions of the virtual resource manager to exploit the capabilities of the 6150 hardware. It provides us with a general base on which to provide support for additional devices, applications, and communications features without massive re-coding or user inconvenience.

Restarting Processes from Core Images in UNIX[†] 4.2BSD

Marco Mercinelli

Sezione Metodologie Software
Divisione Informatica
Centro Studi e Laboratori Telecomunicazioni (CSELT)
10148 Torino, Italy

Riccardo Gusella

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

Extended Abstract

The main objective of this project was to develop a mechanism for saving the image of a UNIX process, i.e. its state and its execution environment, so to be able, at a later time, even after system crashes or across system reboots, to restart its execution. We have called this information, collectively, a *snapshot*. Taking snapshots does not modify in any way the behavior of a process which continues its execution as though nothing has occurred. A process can be restarted from a snapshot at any time, and its execution continues transparently from the point where the snapshot was taken. All process's resources are available and, if necessary, are set to an appropriate state. The snapshot facility can be used as a building block for several higher level mechanisms such as crash recovery, debugging, execution backtracking, and process migration.

A snapshot is a copy of the process execution environment generated by the system when one of two new signals is received by the process. It includes a memory image, the register values, the status of the open files, current directory, and, in general, all the kernel data related to the process. The amount of information needed to form a complete snapshot is not always small or easy to collect. Moreover, there are relationships between processes and their environment that generally cannot be inferred from system information alone. For instance, programs such as *uucp* or *tip* synchronize their access to some resources by creating lock files. There is no explicit or easily deductible relationship between these files and those processes even if they are an essential part of the execution environment.

We anticipate that it is not possible to build a general snapshot/restart mechanism for all the possible computations that UNIX processes can carry out. We have

This work was sponsored by the CSELT Corporation and by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of CSELT, of the Defense Research Projects Agency, or of the US Government.

[†] UNIX is a Trademark of AT&T Bell Laboratories.

therefore to make some assumptions to characterize the class of processes that our mechanism can manage. The first assumption states that the kernel data structures must uniquely determine the execution environment of a process. The second assumption is that the computation carried out by the processes must not be time dependent (i.e. it can be restarted at any later time without logical timing problems). As a further limitation, in the current implementation, we can only save the image of a single process rather than those of a multiprocess computation.

We have implemented all the functions of the snapshot and restart mechanisms in the kernel. We decided to divide the saved information in two files: a *core* file and a file we called *snap*. The core file comprises a copy of the User area, the kernel stack, user stack, and data area of the process. This file has a standard format and can be examined using available debugging tools such as *adb*, *sdb*, or *dbx*. The snap file contains the data not recorded in the core file: process table entry, kernel data describing open files and devices, information to reset the tty state, snapshot date, etc.

The snapshot is produced when a process receives a particular signal. We have introduced two new signals to give the user a minimal control over the snapshot mechanism: *P_SNAP*, and *F_SNAP*. While P_SNAP saves only the process image (in the core and snap files), F_SNAP makes also copies of the open files used by the process in case they need to be modified by the original process when its execution, after the snapshot, resumes. The use of the signal mechanism has several drawbacks. For instance, if a process is sleeping on a high priority event (or if it is stopped), it is not immediately awakened by incoming signals and the snapshot request is served only when the process restarts its execution. Another unsolved problem with signals is that there is no means of returning a code to the signaling process with information on the correct completion of the requested action. However, the use of signals in our implementation allows us to restrict the set of locations inside the kernel where a snapshot can be taken and has greatly simplified the programming effort.

A new system call has been introduced to restore the execution of a process from a saved snapshot. The system call takes as arguments the names of the object, core and snap files, a few control flags, and the address of a buffer where detailed information about error conditions can be returned. The *restart* system call begins by making a *skeleton* of the restarted process: it initializes a process table entry, generates a User area, allocates virtual memory resources and space in the swapping area. The skeleton process is initialized to reflect the execution environment of the original process as it was at snapshot time. When trying to restore the process identifier (pid), restart can find that it is being used by some other process. In this case, the default action is to look for another available identifier. If a process uses its pid value —for example to generate temporary file names— the user can force the system call to allocate the requested pid and return an error in case it is not available.

Restart then restores the current directory, opens the used files and sets the file pointers to the proper position. It then sets the control tty of the new process to the control tty of the calling process. Subsequently, it restores the user data segment and stack from the core file, inserts the process in the running queue for scheduling, and in the end is able to give life again to the original process.

Restart performs many checks during all its phases to avoid meaningless situations. If a check fails, the system call undoes all its work and returns with an error condition. A first group of checks is intended to ensure the consistency of the information of all the snapshot files. Checks are performed on the inodes of the current directory and open files that are to be restored. Each inodes must refer to an existing object of the same type of the original one (directory, file or device) and must be accessible using the current access rights of the restarting process. Dimension and date checks should ensure that files have not been modified. All these checks are not intended for security reasons because they are based on the contents of files that experienced people can always change still keeping a consistent situation. Security is based on standard UNIX mechanisms: restarted processes gain user and group identity from their parent processes, set user and group id's are not allowed and files are accessed according to their new access rights.

We have been able, in an experiment involving two workstations running UNIX 4.2BSD and a common file system, to take a snapshot of a shell process in one workstation and restart it in the other.

A multilingual office automation solution on UNIX/MS-DOS environment.

Roberto Novarese
Giuseppe Pampararo
Giuliano Perego
Stefano Tagliaferri

(Olivetti Software Products Division - Ivrea Italy)

Abstract

In this paper a set of Office Automation requirements of the Economic European Community is presented. A solution that has been designed and prototyped by Olivetti is then discussed. The proposal is based on the integration of UNIX mini and MS-DOS personal computers on a local area network.

Personal Computers Network Support Services provide the sharing of resources over the LAN (file server, print server and communication server functionalities).

A set of integrated Office Productivity Tools on the workstations provide the support to professional and secretarial activities.

An MHS/X.400 Electronic Mail and an Archiving System are the Office Cooperation Services designed to provide user interaction and cooperation and to integrate this solution in a Multivendor System architecture.

UNIX is a trademark of AT&T Corporation.

MS/DOS is a trademark of Microsoft Corporation.

Ethernet is a trademark of XEROX Corporation.

PC Interface is a trademark of AT&T and Locus Corporations.

MS/NET is a trademark of Microsoft Corporation.

1. Introduction

Office automation is perhaps the area in which information technology will most impact the organization of the work in the next future. All computer vendors see office automation how a good business chance, so they are doing high investiments to develop new technologies and products in this field.

In Europe the Economic European Community (EEC) is urging and helping European manufactures on the development of new solution in this area, based on international standards. Besides it needs, as every large organization, an office automation environment allowing creation, handling and integration of documents written in different languages and the distribution to different seats. Therefore EEC has given out a set of office automation requirements to reach the above-mentioned objectives.

Olivetti is interested in both aspects of EEC policy. Infact, as a manifacturer, it is trying to be competitive on the computer market. Moreover it is interested to EEC as the probably most important and innovative office automation European customer.

This paper is divided in five chapters. In the second we give a description of the EEC requirements. In the third we describe the architecture proposed to fulfill them. In the fourth we describe a prototype designed and implemented to meet these requirements. In the fifth the foreseen evolutions of the proposal are discussed.

2. EEC Requirements Description

EEC is organized on different organisms, ranging from political (Parliament, Minister Council) to administrative, technical and economical. These organisms are spread on different geografical sites (mainly Bruxelles and Luxembourg) and have different goals.

As every large organization, EEC has to manage a large amount of structured information (data) and amount of unstructured information (documents).

According to these features its approach to the definition of an office automation project emphatizes the need of integration of document and data, and the problem of information distribuition.

Besides these general requirements, that are common to all large organization, there are two specific of the Community:

i. EEC is a multinational (and therefore multilingual) organization. All the languages of the member countries are official languages of the Community. Every computing equipment used for the production of documents in the community should be able to deal with the different languages and character set. In particular word processing equipment is requested to be able to produce "multilingual documents".

ii. EEC organisms are defining and promoting international "de iure" standards. EEC requires that computing equipment to be sold to its organisms conforms to these standard, and it practices a multivendor policy on equipment acquisition, relying on the ISO-OSI model and protocols for the integration of different systems.

Proprietary solution can he adopted only on a "per product" basis and have to be justified.

When "de iure" standards are not available the EEC policy is to conform to "de facto" standards, on which multivendor support is however guaranteed.

CCITT has given out standard raccomandations about communication and distribuited application packages; in particular EEC requires to adopt X.25 raccomandation for geographic network, the IEEE 802 family of standards for LAN's and X.400 raccomandation for message handling systems.

UNIX can be considered a standard de facto operating system for mini computers, so EEC requires UNIX System V on mini computers as program development environment, while MS/DOS has been choosen as reference OS for intelligent workstation.

The office automation proposal we will describe in the next chapter conforms to these standards, adopting an extension to MS/DOS functionalities to support the multilingual requirement.

3. Architecture Description

The architetture proposed to EEC to cope with its requirement, providing an efficient solution, is based on the distributed client-server model.

With a client-server model the client is a service user and the server is a service provider.

A distributed client-server system is composed by a set of clients and servers distributed across a number of nodes connected through a network.

This architecture is more efficient for office automation applications than a traditional one based on a centralized, powerful system connecting dumb terminals, as it exploits the high interactivity of IWS. Infact while in the classic architecture dumb terminals use the mini's CPU for every operation, IWSs can manage locally a great number of operations related with the interaction with the user, providing support for an higher quality user interface (grafic screen, pointing devices etc.), and managing interaction and cooperation among users on the server shared resources.

Moreover the modularity of the distribuited client-server model allows to fit the structure of the office where the system has to be installed, and guarantees an easy modular extendibility of the system.

From the software point of view, the proposed solution is based on three families of services:

i. Personal Computer Network Support (PCNS's) services. They provide the system-level services to IWS: shared MS/DOS file server disks, spooled access to server printers, UNIX session via terminal emulation, file locking, user autentication etc.

ii. Personal Office Productivity Tools (OPT's). They provide support to individual office activities, using the PCNS services. They include word processing, spreadsheet, agenda, personal database etc. The multilingual requirement have to be satisfied mainly from these packets.

iii. Office Cooperation Services (OCS's). They support structured interaction among system users and allows the exchange of document, messages and data internally to a cluster (a logical group of clients-servers) and with other clusters reacheble via the ISO/OSI communication services , both on LAN links or geographic connections. They include MHS/X.400 Mail Services, Archive Services and access to organizational level computing resorces.

4. Proposed solution and prototype description

In our proposal the client IWSs chosen are machines of Olivetti PC family (M19, M22, M24, M24SP, M28), providing a set of different configurations in terms of computing power and cost with complete compatibility at operating system interface level (MS/DOS). Specific hardware devices has been implemented to fulfill multilingual requirements. In particular special keyboards and printers, able to support the various character sets required from EEC have been designed.

The servers, conforming to EEC requirement of standard OS interface and satisfying the need of multitasking support, have been implemented on 3B computers running UNIX System V.

The LAN on which the systems are connected is Ethernet. In fact, also if this solution is not optimized in terms of per-connection costs, it is the only one guaranteeing multivendor support today.

To provide PCNS services on the prototype the PC Interface package has been selected. This package gives to PC's connected to 3B/UNIX computers the possibility of use them as file servers (mapping the UNIX file system of one of the server as an additional MS/DOS drive) and print servers. The user on the PC logon to the server to which he want gain access to shared data, among the network connected ones. An implicit locking algorithm is implemented to control concurrent access to files, and UNIX protection are enforced on the PC operations. A terminal emulator service allows the PC to become a terminal of the UNIX server to use all sofware packages available on it.

The Olivetti integrated family of OPT's support Office activities running on IWS and accessing data on the server. In particular, in the prototype implementation, some of these packages have been modified to support multilingual documents, providing the capability to dynamically change character set for screen display and keyboard layout. Special printer drivers have been developed.

An extention to MS/DOS screen/keyboard services has been designed and implemented. Sets of languages having the same character set have been identified. A colour has been associated to each of them:

white	Italian, French;
yellow	German, English, Dutch;
green	Spanish;
red	Danish;
violet	Portuguese;
blue	Greek;

to each country colour corresponds a keyboard layout and a character set. A special keyboard with the different layout written on keytops in different colours has been adopted.

In the Olivetti OPT packages there are built-in commands allowing to change colour during a session of work.

A new MS/DOS command selecting a colour allows to select a country char set/keyboard layout before invoking a standard MS/DOS package, so providing general multilingual support.

To implement this "Colour service" a new character generator allowing the management on Olivetti PC's of two character sets has been developed: the standard IBM PC character set and the European one, defined with EEC.

A mail package, conforming to MHS/X.400 CCITT raccomandation supports inter User Personal Messaging and a Document Distribution Service . This package also provide a Directory Service providing mail addresses-name mapping for the users of all the mail connected systems. In the prototype the package is runnig on the 3B/UNIX machine and must be accessed via terminal emulation, and linking of clusters of users (subdomains in MHS terminology) over ISO/OSI communication links is not yet supported. Porting of the User Agent on the MS/DOS IWS environment is foreseen.

No support to archiving activities is provided by the current prototype. Work on this is outgoing and is object of a specific paper ("Office Data Based Services in a UNIX architecture") presented in this session.

5. Evolutions

Respect to the prototype three evolutions are foreseen on the system:

1) The PC Interface will be substituted by MS/NET. Current implementation of PCNS services using the PC Interface present some limitation: it is working on the release 2.11 of MS/DOS, that does not have any "hook" at OS level for networking; it gives to a client access to only one server for session; the access control is implicit and may generate deadlock situation on the net. MS/NET is becoming a standard de facto for PC networking, and the NETBIOS interface is a good base to implement OCS distributed applications using program to program communication.

2) The actual MS/DOS "Colour service" implementation only allows the selection of a country colour before invoking an MS/DOS package. To change language during a session of work, it is necessary to exit from the package, change colour and start the package again. A new mecchanism to change colour will be introduced. With this new tools it will be possible dynamically change language pressing a key combination (eg. CTRL-SHIFT-F1), without exit from the package. The new implementation will use popup menus to interface the user. With this new tool the built-in colour command of Europe versions of OPT's will become unuseful: if it is possible language without stopping the job all standard MS/DOS packages can be considered multilingual. So the OPT offered to EEC will become the standard ones.

3) Last evolution will be in the OCS design. They will allow to access to distributed service without using the host terminal emulator, running the user interface on the IWS's. Each of the package will consist of an user agent working on the MS/DOS IWS and a shared server running on the UNIX machine. The two parts will use the NETBIOS interface to communicate. Beside to the mailing functionalities archiving support will be implemented to manage sharing of data in a controlled way.

All these evolutions will be available for the end of '86. At this date the installation of several system, interconnected on an X.25 network is planned.

An Examination of UNIX standards.

P. J. Peake
UUCP: mcvax!axis!philip
Axis Digital, Paris

ABSTRACT

As UNIX becomes more firmly established in the commercial computing world there is much pressure, and resulting action for standardisation. For example, the SVID and X/OPEN publications.

This presentation looks at some of the problems encountered during the development, and subsequent porting of applications which either run on, or communicate with UNIX systems. Some attention is also paid to the existing standards; as found in practice, and as proposed in the above documents.

UNIX and standards

What is "standard UNIX" ? is a non-trivial question. Since UNIX is such an "open" system many people have produced their own flavours, which are maybe the best thing since sliced bread for them, but give applications programs writers nightmares. There has been a long standing tradition amongst computer manufacturers that proprietry operating systems should be as incompatible as possible, so as to keep a captive clientel. So, cooperation between them to attempt to standardise an operating system interface is relatively new. There have, of course, been co-operative ventures before in the area of programming languages, and one hopes that the results of the current exercise will be better[1].

Unfortunately, some of the basic standardisation questions seem to have been overlooked, and others carefully ignored. This paper will attempt to expose some of the problems we at Axis Digital have found in attempting to work with a wide range of UNIX systems.

The first basic problem with any attempt to standardise UNIX, is that of the existence of two de-facto standards, System 5 and BSD 4.2. This problem is slowly resolving, with the adoption of System 5 by most manufacturers (at least in Europe), and with the decision by DEC to move its ULTRIX system towards full System 5 compatibility.

The human interface

The first point of contact with a computing system is its human interface. This is true for normal users, and for a certain category of programs, for example one of our own products which is a PC/UNIX networking system. Perhaps the most obvious problem in this area is that the existing standards are religously avoided! For example, there are standard line editing commands (# and @, remember them ?), I know of no-one in the whole world who actually uses these editing characters[2].

[1] How many implementations of a given language do you know of that are pure implementations, without extentions and restrictions - both of which make any resulting programs written using them non-portable.

[2] I once posed the question "who uses # and @" to a group of people, the only response I received was "maybe Brian Kernighan, on his teletype 7". This I can say is false, he uses neither these line editing characters, nor a teletype (I asked him).

Another indication that these standards have been virtually forgotten, is the introduction (from System 3 onwards) of a comment facility in the shell -. if he thought about it, who would use a line editing character to introduce a comment ?

Since this standard is totally ignored[3] one finds virtual anarchy; it is totally unpredictable how to edit a command line. This brings us to the next non-standard standard, that of signal generating characters. The "standard" character to generate an interupt is DEL, however, this is frequently used to DELete a character[4]. This becomes even more confusing when various programs (labelit for example) print messages such as "DEL if wrong", since the interupt character is changable, this is an unreasonable message, it is not difficult to find out what the current interupt character actually is and use this in the message (get the hint ?).

So we have the situation where we have some well defined standards for user control characters, which are universally ignored, and anyone sitting before a UNIX terminal is reduced to experimentation to find out what key does what (or, experimentation with the options to stty, to find one which prints what they actually are). For a remote system which must use this interface, trying to write sensible programs to deal with this is non-trivial.

Another small problem for any remote system is that of attempting to clean up after a careless user, or after a communications break, to put an I/O channel into a known state. UNIX lacks what could be described as a "definate kill key". How do you ensure that a program is terminated ? Vi is a good example of a non co-operative program. It refuses to go away upon recepit of an interrupt. It refuses to go away when it reads end-of-file. It even refuses to go away if you send it a quit signal! The only method which (almost) always works, is to generate a hangup signal. However, there is no standard way to do this. The use of the BREAK key is ok if you have a connection which can support the transmission of this non-ascii indication. Even if such a possibility exists, there are (so called) standard System 5 implementations which, due to hardware peculiarities, or to strange bugs in the tty driver don't correctly handle BREAK[5].

Thus as far as the "human interface" of UNIX is concerned, we do have some well defined standards, which are so totally ignored that we may as well not have them. The choice of replacement keys seems to be culture dependent, the table below gives observed usage (some people may argue about the characters used in France, because many sites do actually use the american conventions):

[3] With one exception - in getty, so you have one set of line editing commands at login, and another after! (wierd!).

[4] To my mind, this is more reasonable than backspace, since backspace is (in a way) a printable character. For those people having to deal with strange languages (such as french), a simple and effective method of putting accents onto characters is to use backspace, eg. '<bs>e. Try doing this if <bs> generates an interupt! (no, I don't want to backslash it!)

[5] There is a machine which comes to mind which can't differentiate between a BREAK and a NULL character! I will spare the blushes by not saying which - I just hope that the manufacturer concerned changes his mind about his stated intention of not fixing this bug.

country	erase	kill	eof	intr	quit
England	DEL	cntrl-u	cntrl-z	cntrl-c	cntrl-¦
France	DEL	cntrl-u	cntrl-d	cntrl-c	cntrl-¦
U.S.A.	cntrl-h	cntrl-u	cntrl-d	DEL	cntrl-¦

Another cultural difference is that of shells. There are several so-called System 5's around which insist on using the C-shell. In one case (at least), all of the administration shell scripts have been written for C-shell. Although I have no objections whatsoever to individual users using a shell of their preference, for the sake of uniformity, the default should be the 'Bourne shell', as this is available on virtually any UNIX system. The adoption of such a standard would certainly ease writing of scripts associated with installation and administration of application systems.

The next problem for the poor user who has actually penetrated far enough into a strange UNIX system to be able to accurately type a command line, which is in legal syntax for whatever shell he finds lurking behind the prompt is that of command names, and particularly command options.

I suppose that at this stage I must admit to subscribing to the 'cat -v is bad for you' philosophy. So, I was not terribly amused to find the -v option on the System 5 cat. Nor by the explosion of options on many of the common utilities[6].

Another example of strange ideas which contravene the unstated standard that a command should do only one job, and concentrate on doing that well, is System 5.2 'who'. Who in their right mind would think of using 'who' to find out when the system was last booted ? There is a little logic in this option, but it is obscure enough to not allow 'who' to be the program which would immediately spring to mind as a means of finding this sort of information.

There are many other peculiarities in various systems, which, normally would not be enough to place an insuperable barrier before a user unfamiliar with any given system, but would certainly make his life difficult.

The program environment

For anyone wanting to write applications which must run on any UNIX system, there are even more problems than for the user. The first of these is obviously the C compiler. The problems one finds here vary from C compilers which just generate wrong code, to those who's implementors seem to have found delight in taking the 'Kernighan and Richie' definition and applying the most bizzare (but legal) interpretations possible, wherever possible. The only thing one can say on this score is that to hope to write portable C avoid at all costs anything remotely 'clever', such as structure assignments, variable length argument lists, register variables, enumerated types, identifiers nnon-unique within the first 6 characters, nested include files, pointer coercions etc.

[6] Take 'ls' for example, on System 5.2 an illegal option (if you can find one) gives the following:

```
usage: ls -RadCxmnlogrtucpFbqisf [files]
```

There is a current theory that you can tell a real UNIX guru by the fact that his name is a legal option to ls, unfortunately mine isn't, but, 'ls -alastair', for example, is quite legal.

A particularly annoying trick is that of changing 'lint' to agree with your C compiler. I have found to my cost that spending a long time making an application 'lint free' does not guarantee that you will not have several pages of complaints from lint on another system. One can only hope that the eventual acceptance of the ANSI C standard will solve most of these problems[7].

Having written something which is acceptable to most reasonable C compilers, we are faced with linking the bits together. One of the most common problems is that of C library routines, and the change from terminal ioctl calls to the 'termio' interface. rather that write new ones more in keeping with the last quater of the There are some System 3/5 machines on which the only changes which seem to have been actually made to the V7 source is changing the name of index() to strchr(). Then there are the System 5's which don't have strchr() but do have index()! It is difficult to write programs with 'ifdefs' for different systems when one finds something like a V7 tty control interface, combined with a modified C library.

Assuming that the problem of actual object names and semantics has been sorted out, the next problem is that of loaders. There are some systems around which are half-way attempts to change from a V7 type environment to a System 5. The implementors have not wanted the effort of putting up a completely new compilation system (COFF), so, instead have managed to compile some bits of System 5 with their existing compiler. Ok, but, System 3/5 doesn't give you 'ranlib', and for some reason (maybe its too much of a givaway ?), this is not added. One usually finds no problems loading the standard libraries, because someone has spent lots of time ordering them within the archive. Applications libraries are another matter. One must sometimes give the library to the loader three or four times to resolve all the links. It would probably be possible to perform an ordering exercise on the application libraries, but this is not acceptable, such problems should have dissapeared from the face of the earth by now. The only possible explanation for users of a system to be faced with such problems is either laziness or incompetence on the part of whoever produced the system[8].

Once the program is linked, there is always the possibility that some runtime error will give us a core dump (or maybe just crash the machine!). It can be particularly annoying to be faced with this situation, and to find that neither 'adb' nor 'sdb' exist. (Or that they do exist, but refuse to work; this may be for many reasons, but the most annoying is when 'adb' itself core dumps!) There are some machines on which the debugger is proprietary. This may be nice for existing clients, who, are used to this particular debugging system, but it is next to useless for the rest of the world who have no inclination or intention to learn to use another debugger. Particularly when the only reason you have to use it is because a program which has worked on dozens of other UNIX machines, and which compiles without complaint, core dumps on this one! One of the reasons for the popularity of UNIX is that experience gained is never redundant. The user/programmer is isolated from the idiosyncracies of any given manufacturer. This principle is violated in the case where no standard debugger is supplied. It is quite possible that the proprietary debugging system is better than adb (not

[7] And introduce a whole host of new ones, simply for the sake of allowing a few manufacturers to continue to use their disgusting link-editors rather that write new ones more in keeping with the last quarter of the 20th century.

[8] I would advise anyone finding such a system not to buy it. It may be cheap, but this sort of problem should be taken as an indication of just how well put together the rest of the system is.

difficult), or sdb, but the point is that one or the other of these should be available for users who have a transitory need for a debugger, without the time to learn a new one. By all means throw your super system into the UNIX tools basket, but don't do it at the expense of existing items.

The next problem to be encountered is usually that of maximum fixed sizes imposed by the system on some part of the executable image. This is (almost) always caused by hardware deficiencies in the machine architecture. In this case there is little that can be done, except to search hopefully through the manual pages for the 'cc' entry[9] hoping to find that there is some 'overlay' feature available. Sometimes this will exist, work and solve the problem. Such occasions are rare, as the most common problem is that of limited space allocated to the stack.

What to do about such problems is difficult to say. Different size address spaces for various UNIX systems have always existed. Education of the user comunity is perhaps the best bet.

Exchange media

The means of transfering data onto a machine can be incredibly complicated. The profusion of peripheral devices available makes it most unlikely that when you want to put your sources onto someone elses machine, that you will have a common peripheral. We are quite often reduced to using a UUCP link. This technique is surprisingly effective, slow, yes, but it usually works. The various UUCP versions could themselves do with some standardisation, but it is usually possible to get a link working.

It would be possible to continue to list the potential problems one can encounter with different systems, even when advertised as 'System 5', or 'Derived from AT&T System 5.2 source code'. But rather than continue this depressing story, lets look at what is being done to solve these problems.

The official standards

The term "official standards" is perhaps not quite correct, since neither the SVID[10] nor the X/OPEN portability guide have any real "official" standing. Perhaps the SVID is somewhat more credible since it originates from the "home" of UNIX.

These definitions were awaited with some impatience, perhaps they will resolve our portability problems, we thought. When the SVID first arrived, the first impression was of dissapointment. The colour scheme of the cover seems to have been deliberately chosen to promote feelings of confusion in the observer. Upon opening the book one is immediately struck by the awful quality of the typesetting. The bold print having been produced by double printing with a slight offset. This may just have worked if they had chosen a font with thicker verticals. The actual effect is to enhance the feeling of nausea caused by the cover, as your eyes struggle to accept the "double vision" effect of the typeface.

[9] If you have this sort of problem, it is virtually certain that the system will not have an on-line manual, and that the paper manual is either 'still at the printers', or is an unmodified copy of the AT&T PDP11 compiler page.

[10] System 5 Interface Definition

SVID contents

The contents of the SVID are mostly the System 5 manual pages, with a little bit of re-working. The most apparent change is that the familiar sections of the manual have dissapeared, giving way to sections called OS (Operating System Services), LIB (Libraries) and KEXT (Kernel Extentions). This is somewhat of a mistake. Firstly, it makes life difficult when one tries to find some specific entry. More importantly, some items which are actually library routines are marked as "Operating System Services". It is impossible to distinguish a system call interface from a library routine. The only thing it is possible to say, is that anything marked (OS) will eventually make some system call. However, not all system calls are actually mentioned. There is no mention of sbrk(), but, there is of malloc(), which is marked as (OS). This gives the interesting possibility of some innocent programmer writing something which uses malloc(), and has some routine called sbrk(). When he runs this program there will be some decidedly interesting results, probably terminating in a crash inside a "System Service".

There are errors in this 'definition', some are due to typographical errors, others due to who knows what ? One I find particularly annoying is the (pseudo) BNF definition of a pathname. This is copied directly from the System 5 manual, with the exception that some bright spark noticed that this did not take account of the directory entries '.' and '..'; unfortunately, the modified definition doesn't work either! Just try parsing any pathname containing a '.' or '..' eg. ./fred.

As well as errors there are some inconsistencies. One is informed several times that one should not use 'raw' system calls such as open(), close(), read() and write(), but rather use the stdio functions fopen(), fclose() etc. However, in the section on flock(), hidden away near the end is the admission that you do actually have to use the low level calls if you want the file locking to work correctly. What is not explained is that all programs which are to be used to manipulate a potentialy locked file must avoid using stdio. This lets out all of the standard utilities. If we are going to only be able to use the file locking mechanism in specifically written, dedicated sets of programs, do we actually need file locking anyway ? It only seems to be needed because there exists a breed of programer who think that data-base systems are impossible to write unless the system does most of the hard work for you, and imposes file locking.

An unfortunate problem with this definition is that if someone were to use it as a basis for designing an application, and then hoping that the result would run on any of the systems carrying the label 'System 5.2', he would be in for a nasty shock. A much better bet (for the moment) would be to simply buy a set of Sytem 5 manuals and work from them.

On the credit side, there is the stated intention to adhere to a set of 'levels' for any item appearing within this definition. This basically means that you can guarantee that the next generation of UNIX will (coming from AT&T) still have all the basics there, and there will be no more nasty surprises like termio.

The X/OPEN Portability Guide

This is produced by a group of (mainly) european manufacturers. It is intended as a specification of what will be supported on a UNIX system coming from any of them. Thus allowing applications to be written with the expectation that the same source code will work on a machine coming from any of the group members.

As a book, it is much better in every respect, decent paper, decent type setting, and above all, retains the familiar UNIX manual sections. As far as contents goes, this is better too. The X/OPEN guide is based upon the SVID, and should continue to follow it. This is a welcome relief. Having two competing standards (and BSD!) would just about finish any hopes of ever having a standardised UNIX system. Another comforting sign is that DEC have now joined the X/OPEN group, and so will bring ULTRIX more into line with System 5.

There are extra sections in the X/OPEN guide, concerning the C language, COBOL, and a C-ISAM interface. The C promises to follow the ANSI standard, and the COBOL and C-ISAM are based upon currently existing products.

The last section of the manual is interesting since it specifies a standard for source interchange; magnetic tape and diskette, with specific formats.

On the negative side there is the relegation of the termio interface to the status of an option. The stated reason is that the machines of some of the members cannot implement this interface. Another disturbing sign is that they copied the AT&T definition of a pathname!

Hopes for the future

The simple fact that people are thinking about standardising UNIX implementations is very encouraging. Just what they are thinking may well be another matter. As pointed out above, there are certain areas in which standardisation could be achieved easily. It just needs to be done. There seems to be too much effort being put into difficult areas, such as international character sets, and IPC mechanisms integrated with networking. These are deserving areas for research, but more progress could be achieved, more quickly on simple things, like re-defining the standard line editing and interrupt key conventions, a standard and coherent set of command line options to the standard set of utilities etc.

Perhaps the most encouraging thing is that even in the face of all the difficulties mentioned above, it is actually possible to work on, and to transfer applications to machines of very diverse architecture and manufacture, thanks to UNIX.

The Eighth Edition Unix* Connection Service

David L Presotto

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Using the connection service, processes can connect to processes on the same system or across a variety of networks. Unlike other solutions to this problem, such as 4.2 BSD's sockets, ours abstracts network protocols and communication properties to such an extent that application programs using it need not be cognizant of the network or network protocol on which the connection is built.

The connection service is based on Dennis Ritchie's Streams, a mechanism for providing two way byte streams between processes and devices or between processes, and other processes. The unique properties of Streams make the connection service possible. Using Streams we can perform functions such as circuit setup, circuit shutdown, and data stream processing in the kernel, in processes, or even in separate processors as the situation dictates.

PROLOGUE

Like other versions of Unix, our Eighth Edition Unix† has accreted a number of inter-process communications channels; pipes, pseudo terminals, hard wired lines, auto dialers, multiplexed network devices, etc. In implementing these channels, we have been careful to give them all the same program interface. All channels are represented by capabilities called *descriptors*. A program accesses the channel by performing reads and writes of byte sequences to the descriptors. This is the standard Unix interface used for sequential files. Therefore, in most cases, programs need not be aware of what type of object an I/O descriptor refers to. In fact, a channel used only for reads or only for writes looks to a program exactly like a sequential file. This 'transparency' has been very useful in composing processes as filters in a data pipeline and making programs generally plug compatible.

In 1982 Dennis Ritchie added *streams* to Unix.[1] A *stream* is a full-duplex connection between a process and a device or another process. It consists of several linearly connected *processing modules*, and is analogous to a traditional Shell pipeline, except the data flows in both directions. The processing modules are resident in the kernel and can be pushed onto or popped off the stream by a process which holds a descriptor for the stream. Dennis rewrote all character devices, including communications channels, using the stream abstraction.

The stream abstraction extends the 'transparency' of communication channels. Although all our channels have the same interface for reads and writes, some properties, such as error correction and protocol-processing, differ from channel to channel. Before streams, if a channel lacked a needed property, the property was coded into the process using the channel. As a result, programs written for one channel would often not work when applied to a different type of channel. The

* Unix is a trademark of AT&T Bell Laboratories

† The Eighth Edition version of Unix is the system that runs on machines in the Information Sciences Research Division of AT&T Bell Laboratories.

streams abstraction solves this problem by encoding channel properties into processing modules. Applications programs may apply properties to channels by pushing processing modules which implement these properties onto the ends of the stream representing the channel.

Once a process has obtained a descriptor to a connected channel, with appropriate processing modules attached, it can use the channel without regard to the network or medium supporting it. Unfortunately, no equivalent abstraction existed for establishment of the connection. Each network has a different syntax for the naming of processes and a unique multi-phase procedure for negotiating the connection.

To solve this problem, Dennis Ritchie and I added three new kernel mechanisms to 8th Edition Unix[2]. They consisted of:

- association of a stream end with any named file
- passing file descriptors through streams
- a processing module that supplies unique connections between processes at each end of the stream

These extensions were provided as building blocks for a network independent connection service for streams. This paper describes the resulting service and the name space it provides.

NAMING STREAMS

For two processes to establish communications over a stream, each must obtain a descriptor to one end of the stream. Related processes may inherit these descriptors from a common ancestor just as the shell passes the two ends of a pipe as output and input to processes in a pipeline. However, unrelated processes need some way to specify or 'name' the stream ends in order to open them. In our model, a process wishing to receive connections, the *callee*, obtains a stream and associates one of its ends with a name. It then listens on the other end for call requests. A process wishing to communicate with the callee, the *caller*, requests a descriptor to the stream end with the agreed upon name and uses it to pass connection requests to the caller. This implies the need for a name space for streams.

In providing the name space we were guided by the following considerations:

- Introduce as little 'new' as possible. If a mechanism already exists for something we wish to do, use it.
- Both local and network names should have the same syntax. That syntax should be as simple as possible[3].
- Naming on many networks is hierarchical. If we are to represent network names, or paths through a number of networks, our name space must support hierarchical names.

Our users were already familiar with a name space consistent with all of the above considerations, that used by the Unix file system. Therefore, we merged our name space with that of the file system. This required that stream ends be allowed as nodes of the file system. The implementation section at the end of the paper describes how this was done.

A *name* consists of a sequence of character strings (elements), each sequence delimited by a '/'. The elements represent a path through the name space. If the name starts with a '/', the path begins at the local root (the root of the local machine). Otherwise, the path begins at the position in the hierarchy that the translating process is connected to. In any path, the element '..' means to move one level of the hierarchy toward the root, unless already at the root. '/../' at the start of a path is equivalent to '/'.

The global name space is constructed by splicing together the file system names spaces of different Unix systems and the name spaces of the networks. We do this by attaching the roots of the network name spaces to leaves of each local file system and by attaching the roots of the local file systems onto leaves of the network name spaces. This creates a directed cyclic graph such as the one represented in Figure 1.

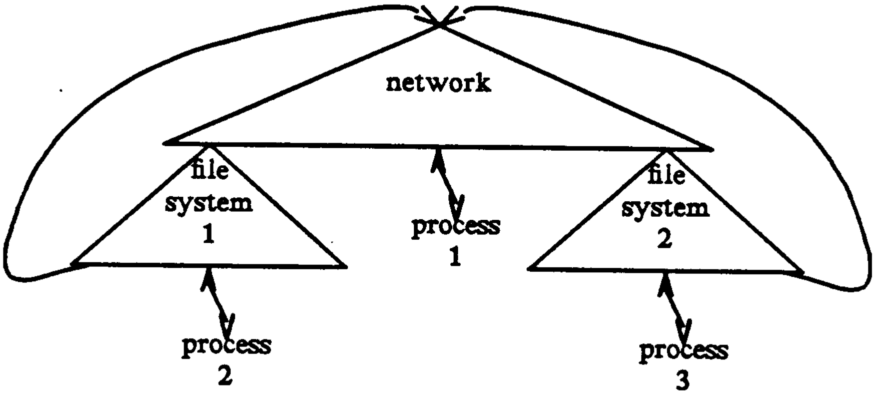


Figure 1 - splicing of file system and network name spaces

This figure depicts two file system name spaces attached to a single network name space. Processes may attach streams to any point in the resultant aggregate name space. In this example processes are attached to each of the three components.

Figure 2 illustrates a concrete example of our name space. Here, the local name spaces of two systems, *research* and *snb*, are depicted. The systems are joined by a Datakit[4] network. The root of the datakit is attached to the node **/dk** of both file systems and both file systems are attached to leaves of the datakit. In addition, 'research' is attached to a TCP/IP network at its leaf node, **/in.** Finally, each system has two local processes attached via streams to the local name spaces at **/login** and **/exec** connected to its name space.

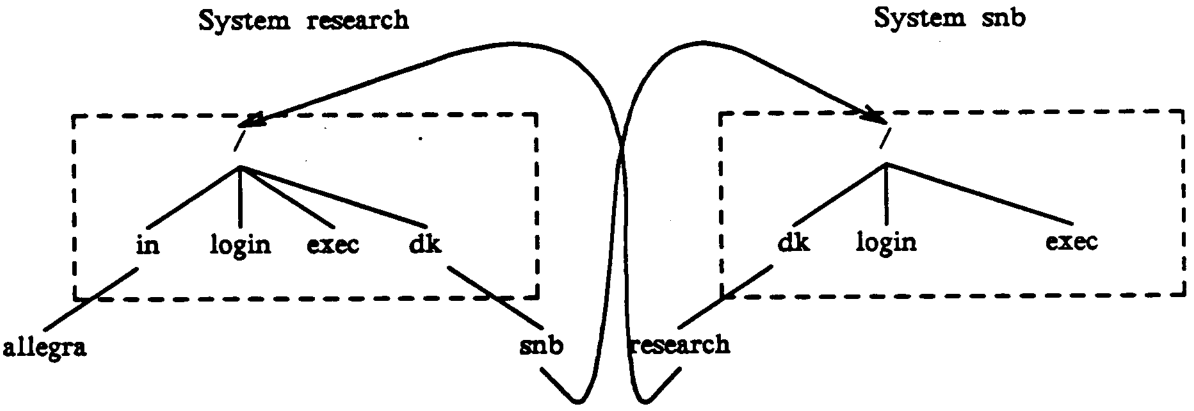


Figure 2 - the name spaces for two Unix systems

We now consider a number of names in this example to see how a process on *snb* might name other processes. The simplest is a local service. For example, to connect to the local execution server, an *snb* process would use the name **/exec**. Similarly, it could use the name **/dk/snb/exec** although this might imply connecting the processes via the network rather than via some faster local communications mechanism. To access the remote login server on *research* the *snb* process would use the name **/dk/research/login.**

One advantage of our naming scheme is the ability to explicitly name a route through a gateway machine. For example, *snb* has no local attachment to the TCP/IP network. However, a process on *snb* may connect to a process on that network by specifying a path through *research* such as **/dk/research/in/allegra.** Such paths greatly extend the name space visible to a process.

PROGRAM INTERFACE

In this section we present the program interface for connection setup. We divide our presentation into two parts, the caller interface and the callee interface. The classification of processes as callers or callees is relative to the establishment of a single connection and doesn't necessarily imply a client/server process model.

On errors, all the library routines described here set two global variables, *errno* and *errstr*. *Errno* is an integer providing a generic reason for the failure. *Errstr* is a character string giving an expanded reason that can presented to the user.

Caller Interface

A caller establishes a unique connection to a callee by using the *iopen* library routine. It then associates with the connection those properties needed for the application with the *iprop* library routine.

The form of an *iopen* call is:

```
    int conn;
    char *name;
    char *props;

    conn = iopen(name, props);
```

where

name is the name of the stream to connect via.

props is a list of connection properties.

conn is the returned descriptor.

Iopen returns a descriptor to a stream representing a new connection or a —1 to indicate an error. The property list *props* is usually empty. However, it may contain a list of properties to be applied to the connection. If any of these properties cannot be applied to the connection, the call fails. Possible properties include *heavy* to indicate heavy traffic expected over the connection, *delimited* to indicate that reads and writes be delimited, and *correct* to guarantee error correction accross the connection.

The caller may subsequently request additional properties to be added to the connection using the *iprop* library routine.

```
int conn;
char *props;

conn = iprop(conn, props);
```

where

conn is the value returned from iopen.

props is a list of channel properties.

The return value is either —1 indicating inability to support the requested properties or a descriptor to the connection. The new descriptor need not be the same as the original one since a process may be inserted between the caller and callee in order to support a requested property. If a new descriptor is returned, the original one is closed and the original value of *conn* becomes invalid.

Since properties may be specified in the *iopen* call, the *iprop* routine may seem redundant. The reason for providing a separate call is to allow more flexibility for the applications program. For example, if a program cannot obtain a channel with the desired properties, it may fall back to a different protocol for the connection. UUCP, for example, has different protocols for different types of channels.

Callee Interface

Four library routines, in addition to the *iprop* routine mentioned above, make up the callee's interface. The first is *icreat*. The callee uses this to attach itself to the name space. The form of the call is:

```
int named;
char *name;
char *param;

named = icreat(name, props);
```

where

name is the name to attach a stream to

props is a list of properties.

named is a descriptor to the stream attached to *name*.

If *icreat* succeeds in attaching the stream to the name space, it returns a descriptor to the other end of the stream. If it couldn't, it returns —1.

The callee then waits for calls using the *ilisten* library routine. This routine returns only when a call request has been received.

```
struct ipcinfo {
        int conn;        /* fd for connection(callee side) */
        char *name;      /* the name of the stream end being called */
        char *machine;/* machine id of caller */
        char *user;      /* user name of caller */
} *ip;

ip = ilisten(named);
```

Ipcinfo is a structure defining the call request. Using the information in *ipcinfo*, the callee chooses whether to accept the call or reject the request. To reject the request:

```
int code;
char *reason;

ireject(ip, code, reason);
```

where

code is the value to be assigned to the caller's *errno*.

reason is the value to be assigned to the caller's *errstr*.

Ireject passes the error information to the caller and closes all descriptors associated with the call. The callee may then continue listening for more requests.

If the callee wishes to accept the call, it first associates the appropriate properties with the communications stream, *ip—>conn*, using the *iprop* routine and accepts the call with *iaccept*.

```
int newconn;
ipcinfo *ip;

ip—>conn = iprop(ip—>conn, param);
iaccept(ip, newconn);
```

Here, *newconn* is a descriptor for a different stream than the one originally supplied in *ip—>conn*. This allows the callee to supply its own stream for the connection. Normally this is —1 indicating no stream supplied. *Iaccept* returns 0 on success and —1 on failure.

To allow the caller to poll a number of named streams without blocking, the *select* system call may be used on the descriptor to the named stream.

IMPLEMENTATION

We present three additions to Unix and then show how they are used to implement the name space and program interface described above.

Additions

We made three additions to the system.

Mounted streams

First is a new, but very simple, file system type. Its *mount* request attaches a stream named by a file descriptor to a file. Most often the stream is one end of a pipe created by the server process, but it can equally well be a connection to a device, or a network connection to a process on another machine. Subsequently, when other processes open and do I/O on that file, their requests refer to the stream attached to the file. The effect is similar to a System V FIFO that has already been opened by a server, but more general: communication is full-duplex, the server can be on another machine, and (because the connection is a stream), intermediate processing modules may be installed.

By itself, a mounted stream shares the most important difficulty of the FIFO; several processes attempting to use it simultaneously must somehow cooperate.

Passing files

The second addition is a way of passing an open file from one process to another across a pipe connection. Although they are actually done with *ioctl* operations, the primitives may be written

sendfile(wpipefd, fd);

in the sender process, and

(fd1, info) = recvfile(rpipefd);

in the receiver. The sender transmits a copy of its file descriptor *fd* over the pipe to the receiver; when the receiver accepts it, it gains a new open file denoted by *fd1*. (Other information, such as the user- and group-id of the sender, is also passed.)

Unique connections

Finally, we found a way for each client of a server to gain a unique, non-multiplexed connection to that server. It takes the form of a processing module that can be pushed on a stream, which will usually be mounted in the file system as described above. When the file is opened by another program, this module creates a new pipe, and sends one end to the server process at the other end of the mounted stream, using the same mechanism as the *sendfile* primitive described above. After the server has called *recvfile* to pick up its end of the pipe, it may accept or reject the new connection; if it accepts, the other program's *open* call succeeds, and its open file refers to the local end of the new pipe to the server. If the server rejects the request, the *open* fails.

Library Routines

icreat

This routine walks the path specified by the argument *name*. If the path terminates in the local file system, it creates a pipe, pushes a 'unique connection' processing module on one end, and attempts to mount that end into the file system. If the attempt succeeds, the descriptor to the other end is returned.

If, however, *icreat* encounters a mounted stream before it reaches the end of the path, this signals the interface between the local file system and a network. The process mounted there is a network dialer and will perform whatever functions are necessary to mount into the networks name space. *Icreat* passes the rest of the name and the property list to the dialer. If the dialer succeeds in establishing the name in the network name space, it passes back a descriptor to a stream associated with the name. *Icreat* returns that stream to its invoker.

iopen

Just like *icreat*, *iopen* walks the path specified by *name*. When it reaches a mounted stream, it opens the stream obtaining a unique connection, the reply stream, to the process at the other end. It then passes the remaining path elements and the property list over the reply stream and waits for a reply. If the call is accepted, a new stream descriptor for the actual connection will be sent over the reply stream. If the call is rejected, the reason is read from the reply stream. *Iopen* then closes the reply stream and returns the descriptor for the new connection or —1 for an error.

ilisten and iaccept

Ilisten reads from the named stream waiting for connection requests in the form of a descriptor for a reply stream. When one is received, it reads from that descriptor the text of the request, creates a pipe to use for the connection, and returns all this information in an *ipcinfo* structure to its invoker.

A subsequent *iaccept* will cause an acceptance message to be sent over the reply stream to the caller. If no connection descriptor is supplied in the *iaccept* invocation, one end of the pipe created by the previous *ilisten* is sent over the reply stream. Otherwise, the supplied descriptor is returned over the reply stream and the pipe created by *ilisten* is closed. The reply stream is then closed.

iprop

After parsing the property list, *iprop* consults a compiled in table for the actions to perform for each property. The table includes the *ioctl* system call to use to determine if the stream possesses the desired property, the 'processing module' to push on the stream to implement the property, and the name of the binary to execute to provide the property. If the property already exists, *iprop* goes on to the next in the list.

If the stream doesn't contain the property, *iprop* first tries pushing the appropriate 'processing module' onto the stream. If this doesn't work, it forks a process to implement the property and returns a descriptor to a pipe to that process.

Network Dialers

Network dialers are processes that connect the network and local name spaces. A network dialer starts by performing an *icreat* to mount itself into the local file system. It also announces itself to the network at the appropriate place in the network name space. It then converts between the network's idea of a call setup and that of the local processes.

When a caller on system A attempts to open a callee on system B, system A's dialer functions as a callee and system B's as a caller. Together, the dialers set up a connection through the network and pass the two ends of the connection to caller and callee.

Dialers are by far the most complicated part of the connection service. They embody all of the network specific code.

STATUS AND CONCLUSIONS

We have built a prototype system and are in the process of converting all our network software to use the interfaces specified here. We have already seen a number of advantages to the unified name space.

- Services that used to be limited to a single network are now generic.
- Since systems can now act as gateways between different protocol families, we no longer need to support every network in every system's kernel.
- Many existing programs are being coalesced.

However, generalization often is bought at the expense of performance. After conversion is completed, we will be comparing performance between the old and new systems. Initial experiences, however, indicate that the performance difference is not significant. Hopefully, we will soon be backing this up with numbers.

REFERENCES

1. D. M. Ritchie, "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal* 63(8) October 1984

2. D. L. Presotto and D. M. Ritchie, "Interprocess Communication in the 8th Edition Unix System", *Proceedings of the 10th Usenix Conference*, June, 1985

3. R. Pike & P. J. Weinberger, "The Hideous Name", *Proceedings of the 10th Usenix Conference*, June, 1985

4. A. G. Fraser, "Datakit - A Modular Network for Synchronous and Asynchronous Traffic", *Proc. Int. Conf. on Commun.*, Boston, MA (June 1980)

Developing Software for a Graphics Terminal in C

J.E. Richards

University of Bristol Computer Centre
University Walk, Bristol, BS8 1TW, U.K.

ABSTRACT

This paper describes the development of software for incorporation in a new intelligent raster graphics terminal. The terminal provides support for windows and graphics segments. The software was written in C and developed and tested on a UNIX™ system before being placed in ROM in the terminal. The paper shows how considerable use was made of structures and the storage allocation functions to provide a generalised segment storage scheme. Examples are given of the way language constructs were used to obtain fast, but portable, code. The finished software was ported to the terminal with hardly any modifications.

1. Introduction

Since the beginning of the eighties, the raster graphics terminal has become the most commonly used graphics device, superceding the storage tube devices (e.g. the Tektronix 4010 and 4014 [1]) of the late 1970's.

Storage tube terminals are only capable of simple graphics functions, such as drawing lines and displaying cross-hair cursors. Raster graphics makes many extra functions possible, such as polygon filling, selective erasure, and the use of colour. However, there are some obstacles to the effective use of raster graphics.

Firstly, the designers of the terminals do not necessarily know what functions the user most want. Features are added in a haphazard fashion: for example, one company's definition of polygon filling can be substantially different from another's. The encoding of commands and coordinates is another area where there is no agreement.

Secondly, the software packages of the user might not support raster graphics or the command set of a particular terminal in a sensible way, or at all. In the case of GINO [2], a graphics subroutine library that is used by a number of British academic institutions as well as commercial companies, a version supporting polygon filling and colour look-up tables did not appear until 1984!

The result is indecision on the part of manufacturers, software developers and end-users about the facilities that are needed and the form in which they should be provided. It is hoped that recent international and ANSI standards activity will lead to an improvement.

The Graphical Kernel System, GKS, has been adopted as an international standard by ISO [3]. It defines a library of graphics functions which can be called from a user's program. The capabilities of raster graphics devices were taken into consideration when the standard was being developed. GKS alleviates one of the above problems by defining rigorously the functions that are available and their effect. However, the GKS standard does not define the interface between GKS and a graphics device. This problem is being considered by ANSI who are developing the Computer Graphics Interface, CGI, standard [4]. CGI will define a set of required and non-required functions for graphics devices and a way of encoding commands and coordinates for transmission between a host computer and a device.

This paper describes part of a project to develop a new graphics terminal providing high-level functions, in the light of the progress on graphics standards. The terminal contains a conventional 16-bit microprocessor with a large amount of memory. The software to control the terminal is contained in EPROM.

The software, called Centaur, is written in the C programming language [5]. All of the development, apart from the final stages of porting the code to the terminal, was done on UNIX™ systems.

The software was constructed in three stages, described in Section 3. The first two stages involved constructing a simulation of the new terminal. In the last stage, the software was ported to the development terminal.

2. Functionality of the Terminal

The selection and design of functions for the terminal were influenced by the GKS standard and the CGI standard proposals. Additionally, attention was given to the set of functions provided on existing terminals, such as the Tektronix 4107 [6]. In particular, the definition of coordinate systems, windows and viewports is based on those of the 4107.

Placing more functions in the device has the advantage of decreasing the amount of processing that has to be performed by the host computer. There is the additional advantage of reducing the number of bytes that have to be sent between the host computer and the terminal over a slow link, such as an RS-232C serial line running at 9600 or 19200 Baud.

The drawing speed of the terminal was an important consideration, so it was decided to use integer variables and integer arithmetic operations wherever possible.

The terminal has the following functions:

2.1 Output Functions

There are four output primitives (see Fig. 1):

Polyline	A sequence of connected lines. Can be drawn in different types.
Polymarker	A set of small symbols, available in several types.
Polygon	Can be drawn hollow, solid, or filled with a pattern.
Text	Several fonts, can be drawn any size, orientation, slant or direction.

2.2 Output Attributes

Each output primitive has a set of *attributes*. The attributes affect the appearance of the primitive on the screen. For example, the polyline primitive has colour index and line type attributes. The text primitive has a total of ten attributes, including character height, width, spacing and slant, text rotation and colour index.

The values of the attributes can be set by the user. However, once a primitive has been created, the attributes of that primitive cannot be changed. This is consistent with the GKS concept of *binding* attributes to primitives.

All output primitives have colour index attributes. The number of colours depends on the number of bit-planes in the terminal, but the software can handle up to 32768 different colours. The minimum number would be a choice of 16 colours from a palette of 4096.

2.3 Windows and Viewports

Coordinates are expressed in a 4096x4096 integer coordinate space. Transformations can be constructed between a *window* in this space to a *viewport* on the screen of the terminal (see Fig. 2). Changing the size of the window affects the scaling of the image. Changing the location of the window causes a *pan* across the image. Several windows and viewports can exist simultaneously. They can be made invisible and can overlap.

2.4 Segments

Graphics output primitives can be stored in *segments*. A segment is physically stored in the memory of the terminal, and can be scaled, rotated or moved to a new position. Segments can also be made invisible, highlighted, copied and deleted.

The provision of segments in the graphics terminal has several benefits. Once output primitives have been stored in a segment, there is no need for them to be retransmitted by the host computer. Subsequent calculations on the segments can be performed by the terminal. For example, segments could contain pictures of electrical components for a printed circuit board layout program. The task of positioning the components can be performed by the terminal with minimal intervention by the host computer.

2.5 Input Functions

There are three input functions:

Locator The coordinates of a point are returned to the host computer when the user presses a key or a button on a mouse.

Stroke A series of points are obtained (such as on a curve drawn free-hand) and returned to the host computer.

Pick The user moves the cursor so that it points to a segment. When a key or button is pressed, the name of the segment is returned to the host computer.

3. Simulating the Terminal

The Centaur software was written and tested before constructing any of the terminal's hardware. This was done by simulating the terminal in three stages. Fig. 3 shows the eventual configuration, with a host computer running a graphics applications package, communicating with a terminal running Centaur.

First, the graphics terminal was simulated by running all the software on the host computer as shown in Fig. 4. The Centaur software ran as a process on the host computer communicating via pipes with a test graphics applications package running as another process. The output from Centaur was sent to a simple graphics terminal, capable of line and polygon drawing and erasure. This scheme enabled evaluation and initial debugging of the software.

However, the simulation ran very slowly and response times were difficult to predict; the host computer was a DEC VAX™ 11/750 running Berkeley UNIX 4.1bsd with up to 32 users. In order to gauge the response and processing times more effectively, a different configuration was adopted (see Fig. 5).

A microcomputer, running UNIX System III, was placed between the host computer and the simple graphics terminal, communicating with each by serial lines. The Centaur process was moved to the microcomputer. As the first simulation had been designed with two separate processes, very little rewriting of code was necessary to move one process to another machine. Communication over a serial line replaced the communication by pipes and the calls to the **read** and **write** functions could be left unchanged.

This configuration gave a much better indication of the likely eventual running speed of the software. The microcomputer had the same processor, a Motorola 68000, as was to be used in the terminal. So, in addition to further debugging of the code, optimization could be carried out at an early stage.

The test graphics applications package was now exchanging bytes with what appeared to be, from the host computer end, an "intelligent" graphics terminal. This meant that it was possible to verify that the workload on the host computer was reduced substantially for certain tasks.

The final step was to port the code to a microprocessor development system, compile it and place it in EPROM. This transfer was made with only minor changes required to the Centaur code, reinforcing the choice of the C language for its portability.

A large amount of code was added at this stage to interface the Centaur functions to the hardware, replace UNIX system calls with appropriate functions, and provide all the other requirements of the terminal, such as the text editing functions. Most of this latter code could have been included in the earlier simulations. Future development projects may attempt to increase the percentage of code covered in the simulations.

4. Segment Storage

One of the requirements of the project was that the terminal should be able to store segments containing graphics primitives [7]. One possible way of doing this is to store the bytes sent from the host computer in a linear array called a *display list*. Extra bytes are added to the list to indicate the beginning and end of each segment. Redrawing all the segments is performed by passing through the list and decoding all the bytes as though they had just been sent by the host computer. This approach has the advantage of simplicity when constructing the list and redrawing segments. However, deletion of segments can be a problem as the list has to be regularly compacted to avoid "holes". Additionally, each segment is given a priority, which determines which segments appear in front of or behind it. Changing a segment's priority usually means altering its position in the list, resulting in considerable problems in reordering the list.

After some consideration, it was decided to use the alternative approach of a linked list for segment storage. One factor in the decision was that C provides a natural way of referring to segments and primitives, by defining structures to contain the data and pointers to link the structures together. Another factor was that UNIX provides storage allocation functions (**malloc** and **free**), which were perfectly adequate for the task of allocating and freeing space for segments and output primitives.

The segment storage is set up as shown in Fig. 6. Segments are stored in the list in order of increasing priority. Redrawing of segments is performed by starting at the First Segment pointer and following the pointers to the output primitives. Then the pointer to the next segment is taken, and so on until the last segment is reached. Thus, segments of low priority are drawn on the screen first and may be covered by segments of higher priority.

A segment is deleted by freeing all the relevant parts of the segment storage and rearranging the pointers of the two neighbouring segments.

Changing the priority of a segment is equivalent to changing its position in the list of segments. With this scheme of segment storage, the segment does not have to be copied; the priority can be changed by rearranging pointers.

It can be seen from Fig. 6 that the list of segments is doubly-linked; for each segment there is a pointer to the next segment and to the previous segment. This is to facilitate the one case, the pick input function, when the list has to be traversed in order of decreasing priority. For the pick function the cursor is used to point to a segment. If two segments are indicated, the name of the one with the highest priority must be returned. In order to avoid checking segments unnecessarily and thus increasing the response time, the segments are inspected in order of decreasing priority. As soon as a segment is found that satisfies the pick operation no further segments need to be examined.

5. Portability

The *lint* utility [8] was used to pinpoint any doubtful constructs. In addition, the type of data items had to be considered carefully. Coordinates are stored as 16-bit integers, but it transpires that it is not sufficient to declare them as of type **short**, as it cannot be guaranteed that **short** items will be the correct length. One C compiler on a non-UNIX system uses **short** to represent a one byte variable! Instead, derived types were used throughout, and the standard types confined to one include file. For example:

```
typedef Int16 short;  /* 16 bit signed integer */
typedef Coord Int16;
Coord    x, y;
```

If the code is moved to a new system, the **typedef Int16** may need to be changed. If, at a later date, it is decided to use 32-bit integers or floating point variables for coordinates, then the **typedef** of **Coord** would need to be changed. The important point is that using **typedefs** improves the portability of the software by restricting necessary changes to one small area of code, and makes it possible to consider more fundamental changes in the future, such as the switch from integer to real coordinates.

6. Efficiency

In a device like a graphics terminal, speed is a paramount consideration, but using a high-level language instead of an assembly language is bound to introduce inefficiencies. The generation of just one output primitive is a complex operation. For example, producing a filled polygon involves the following steps:

a. Clip the polygon to the current window.

b. Transform the polygon's vertices from the user's coordinates to the terminal's coordinates.

c. Fill the polygon.

Additionally, if the polygon is stored in a segment, the segment may have been scaled, rotated and repositioned, requiring another transformation to be applied to all the vertices before they can be clipped. The generation of polylines and polymarkers is simpler, but producing high-quality text can take considerably more processing time.

This process has to be performed for possibly thousands of primitives that can be on the screen simultaneously. Consequently, optimization of the code was considered to be extremely important.

One operation that was found to be taking a considerable amount of time was the transformation from the user's coordinates to the terminal's coordinates. As explained above, the transformation is defined by a window in the user's coordinates and a viewport in the terminal's coordinates. The contents of the window are transformed to appear in the viewport. Fig. 2 shows an example, a window with the lower-left corner at *(xw1, yw1)* and the upper-right corner at *(xw2, yw2)*. The corresponding corners of the viewport are at *(xv1, yv1)* and *(xv2, yv2)*. Just considering the *x*-coordinate for simplicity, a transformation can be constructed by the following code:

```
float scale;
int    wwidth, xw1, xw2, vwidth, xv1, xv2, disp;
    /* Set up transformation */
wwidth = xw2 - xw1;                /* window width */
vwidth = xv2 - xv1;              /* viewport width */
scale = vwidth / (float)wwidth;  /* scaling factor */
disp = xv1 - scale * xw1;        /* displacement */
    /* Then, to transform one x-coordinate */
x = x * scale + disp;
```

The first thing to notice is that the floating point multiplication, **x * scale,** is performed many times, and is very slow as there is no floating point assistance in the hardware. A substantial improvement, over 300%, is obtained by rewriting it as an integer multiplication and division.

```
x = x * vwidth / wwidth + disp;
```

Further optimization is possible. Another 10% can be gained, at the expense of defining **x** as a long, by rewriting the expression as:

```
x *= vwidth;
x /= wwidth;
x += disp;
```

This sort of optimization is only worthwhile for sections of code that are performed many times.

Another such case was found in an idle loop that checks for characters arriving from the host computer. The **nochar** function returns true if there is no character waiting in the input queue. It is important that the terminal responds as soon as a character arrives, so the function must be called as often as possible. The loop was originally written as:

```
while (nochar());
```

Surprisingly, it is more efficient to write this as:

```
do;
while (nochar());
```

The former loop compiles to a test and two branches, and the latter to a test and one branch. The optimizing pass of a C compiler will usually detect this case, but it fails to do so in the case of at least one compiler.

7. Conclusions

The development of software for a new graphics terminal has been described. The UNIX operating system and the C programming language were both used successfully throughout the project.

The use of the UNIX system enabled the software to be simulated on first one and then two computers. This meant that the project could be evaluated in stages before any hardware was constructed. The storage allocation functions provided in the UNIX libraries led to a quick implementation of a graphics segment storage scheme.

The C language was found to be excellent for this type of application. Its low-level features permitted the writing of reasonably efficient code. It was possible to write portable code; Centaur has been ported to a number of systems, including MS-DOS as well as UNIX.

Development of the graphics terminal is proceeding well. It is hoped that a product will be launched later in 1986.

8. Acknowledgements

I would like to gratefully acknowledge the financial support of Data Type Ltd, and to thank Derek Buckle, Richard Evans and Richard Morgan-Bedwell. In particular, I must thank Will Parker who wrote nearly all the code after the simulation stages and found most of my bugs; debugging in assembly language is a dying art. He contributed a number of ideas to this paper.

I would also like to thank Chris Morris, Bob Walker, Steve Fisher and Sheila Roberts of the Computer Centre for providing such a good UNIX service.

References

[1] Tektronix, 4010 and 4010-1 Users' Manual, Tektronix Inc., Beaverton, Oregon, USA (1976).

[2] CADCentre, GINO-F 2.7A User Guide, CADCentre Ltd, Cambridge (October 1983).

[3] ISO, "Graphical Kernel System (GKS) - Functional Description," ISO DIS 7942 (June 1983).

[4] T.N. Reed, "Standardization of the Virtual Device Metafile and the Virtual Device Interface," Computers & Graphics Vol. 9(1) pp. 33-38 (1985).

[5] B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

[6] Tektronix, 4107/4109 Computer Display Terminal Programmers' Reference Manual, Tektronix Inc., Beaverton, Oregon, USA (November 1983).

[7] J. D. Foley and A. van Dam, Fundamentals of Interactive Computer Graphics, pp. 155-167, Addison-Wesley, Reading, Mass, USA (1983).

[8] S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (1978). updated version TM 78-1273-3

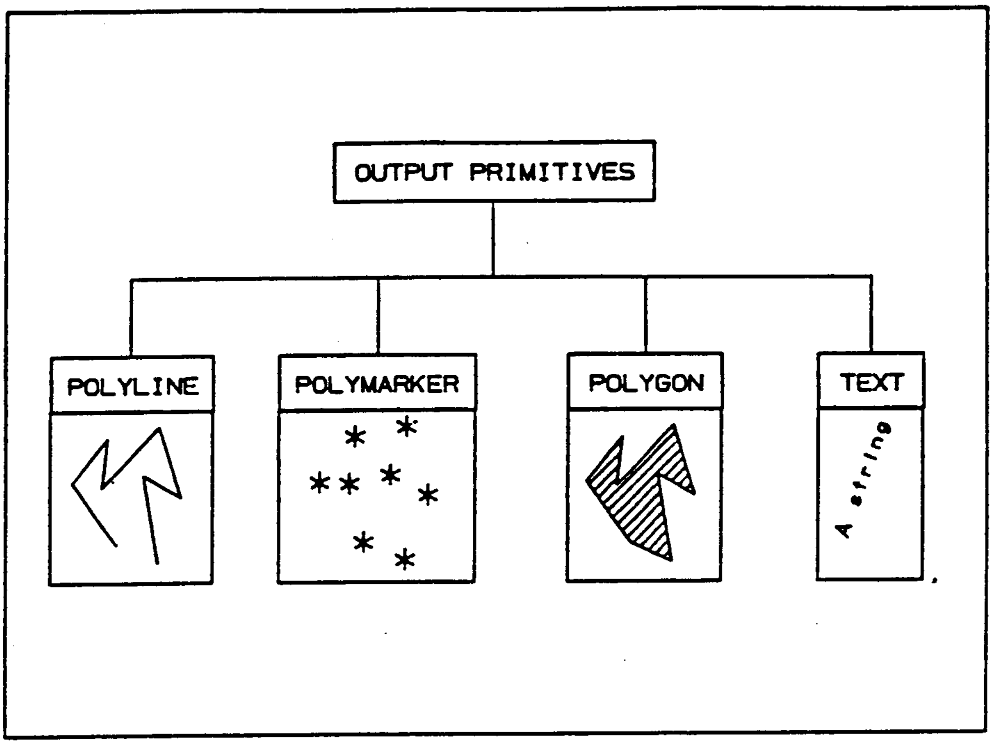


Fig. 1. The output primitives

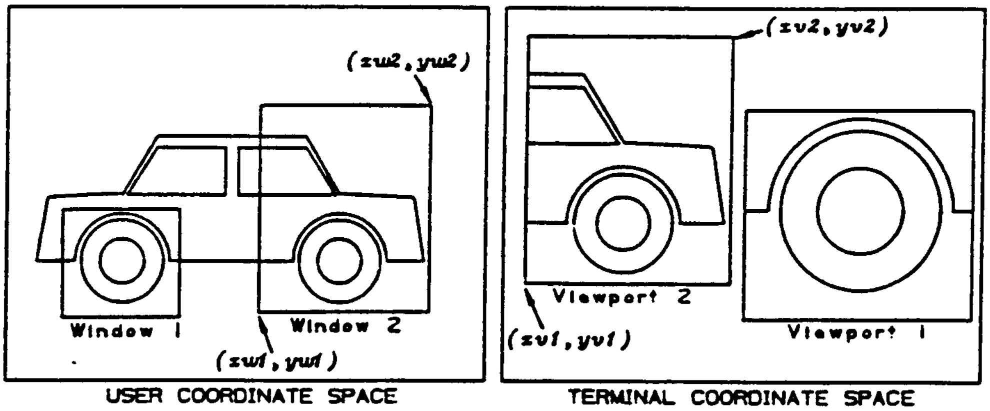


Fig. 2. Transformations from User Coordinates to Terminal Coordinates

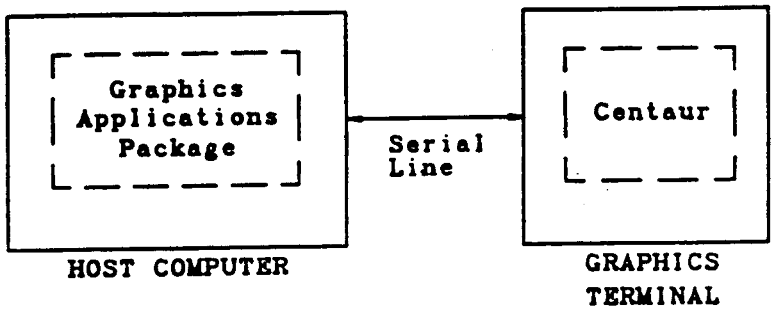


Fig. 3. The final configuration

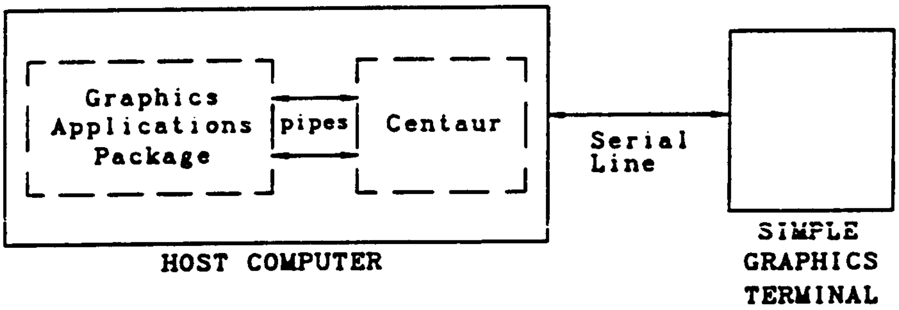


Fig. 4. Simulation 1 - Host running both processes

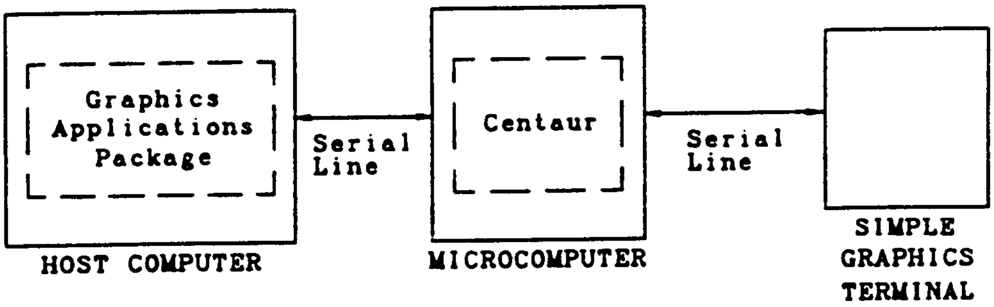


Fig. 5. Simulation 2 - Host running graphics package; micro running Centaur

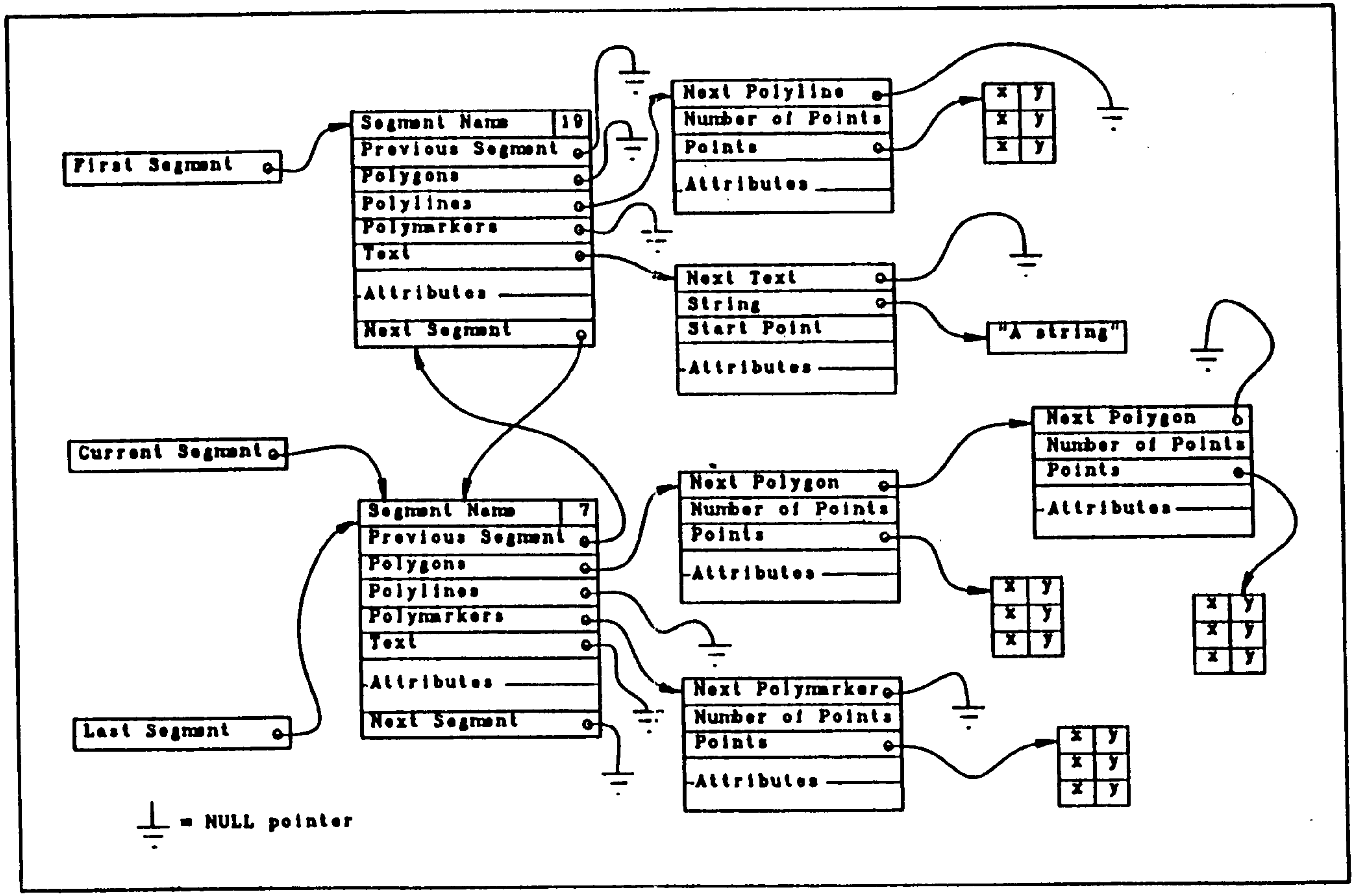
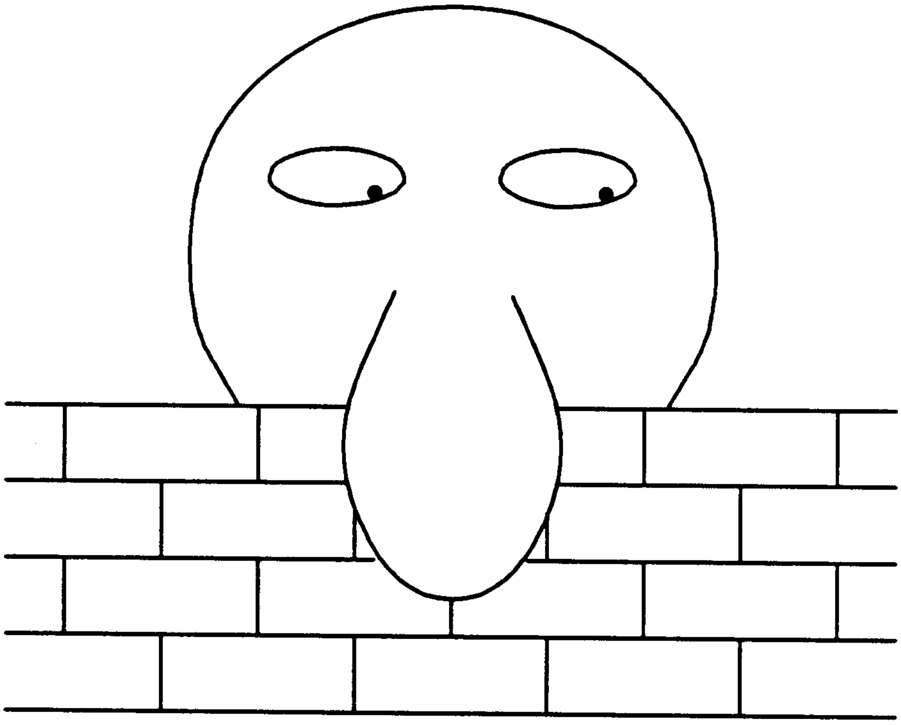


Fig. 6. Layout of the segment storage

RFS in System V.3

Andy Rifkin



"Wot, No paper?"

The Sun Network Filesystem: Design, Implementation and Experience

Russel Sandberg

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA. 94110
(415) 960-7293

Introduction

The Sun Network Filesystem (NFS) provides transparent, remote access to filesystems. Unlike many other remote filesystem implementations under UNIX†, NFS is designed to be easily portable to other operating systems and machine architectures. It uses an External Data Representation (XDR) specification to describe protocols in a machine and system independent way. NFS is implemented on top of a Remote Procedure Call package (RPC) to help simplify protocol definition, implementation, and maintenance.

In order to build NFS into the UNIX kernel in a way that is transparent to applications, we decided to add a new interface to the kernel which separates generic filesystem operations from specific filesystem implementations. The "filesystem interface" consists of two parts: the Virtual File System (VFS) interface defines the operations that can be done on a filesystem, while the virtual node (vnode) interface defines the operations that can be done on a file within that filesystem. This new interface allows us to implement and install new filesystems in much the same way as new device drivers are added to the kernel.

In this paper we discuss the design and implementation of the filesystem interface in the UNIX kernel and the NFS virtual filesystem. We compare NFS to other remote filesystem implementations, and describe some interesting NFS ports that have been done, including the IBM PC implementation under MS/DOS and the VMS server implementation. We also describe the user-level NFS server implementation which allows simple server ports without modification to the underlying operating system. We conclude with some ideas for future enhancements.

In this paper we use the term *server* to refer to a machine that provides resources to the network; a *client* is a machine that accesses resources over the network; a *user* is a person "logged in" at a client; an *application* is a program that executes on a client; and a *workstation* is a client machine that typically supports one user at a time.

Design Goals

NFS was designed to simplify the sharing of filesystem resources in a network of non-homogeneous machines. Our goal was to provide a way of making remote files available to local programs without having to modify, or even relink, those programs. In addition, we wanted remote file access to be comparable in speed to local file access.

The overall design goals of NFS were:

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 clients. The protocols should also be simple enough that they can be implemented on low-end machines like the PC.

Crash Recovery

When clients can mount remote filesystems from many different servers it is very important that clients and servers be able to recover easily from machine crashes and network problems.

Transparent Access

We want to provide a system which allows programs to access remote files in

† UNIX is a trademark of AT&T.

exactly the same way as local files, without special pathname parsing, libraries, or recompiling. Programs should not need or be able to tell whether a file is remote or local.

UNIX Semantics Maintained on UNIX Client
In order for transparent access to work on UNIX machines, UNIX filesystem semantics have to be maintained for remote files.

Reasonable Performance
People will not use a remote filesystem if it is no faster than the existing networking utilities, such as *rcp*, even if it is easier to use. Our design goal was to make NFS as fast as a small local disk on a SCSI interface.

Basic Design

The NFS design consists of three major pieces: the protocol, the server side and the client side.

NFS Protocol

The NFS protocol uses the Sun Remote Procedure Call (RPC) mechanism [1]. For the same reasons that procedure calls simplify programs, RPC helps simplify the definition, organization, and implementation of remote services. The NFS protocol is defined in terms of a set of procedures, their arguments and results, and their effects. Remote procedure calls are synchronous, that is, the client application blocks until the server has completed the call and returned the results. This makes RPC very easy to use and understand because it behaves like a local procedure call.

NFS uses a stateless protocol. The parameters to each procedure call contain all of the information necessary to complete the call, and the server does not keep track of any past requests. This makes crash recovery very easy; when a server crashes, the client resends NFS requests until a response is received, and the server does no crash recovery at all. When a client crashes, no recovery is necessary for either the client or the server.

If state is maintained on the server, on the other hand, recovery is much harder. Both client and server need to reliably detect crashes. The server needs to detect client crashes so that it can discard any state it is holding for the client, and the client must detect server crashes so that it can rebuild the server's state.

A stateless protocol avoids complex crash recovery. If a client just resends requests until a response is received, data will never be lost due to a server crash. In fact, the client cannot tell the difference between a server that has crashed and recovered, and a server that is slow.

Sun's RPC package is designed to be transport independent. New transport protocols, such as ISO and XNS, can be "plugged in" to the RPC implementation without affecting the higher level protocol code (see appendix 3). NFS currently uses the DARPA User Datagram Protocol (UDP) and Internet Protocol (IP) for its transport level. Since UDP is an unreliable datagram protocol, packets can get lost, but because the NFS protocol is stateless and NFS requests are idempotent, the client can recover by retrying the call until the packet gets through.

The most common NFS procedure parameter is a structure called a file handle (fhandle or fh) which is provided by the server and used by the client to reference a file. The fhandle is opaque, that is, the client never looks at the contents of the fhandle, but uses it when operations are done on that file.

An outline of the NFS protocol procedures is given below. For the complete specification see the *Sun Network Filesystem Protocol Specification* [2].

null() returns ()
Do nothing procedure to ping the server and measure round trip time.

lookup(dirfh, name) returns (fh, attr)
Returns a new fhandle and attributes for the named file in a directory.

create(dirfh, name, attr) returns (newfh, attr)
Creates a new file and returns its fhandle and attributes.

remove(dirfh, name) returns (status)
Removes a file from a directory.

getattr(fh) returns (attr)
Returns file attributes. This procedure is like a stat call.

setattr(fh, attr) returns (attr)
Sets the mode, uid, gid, size, access time, and modify time of a file. Setting the size to zero truncates the file.

read(fh, offset, count) returns (attr, data)
Returns up to *count* bytes of data from a file starting *offset* bytes into the file. **read** also returns the attributes of the file.

write(fh, offset, count, data) returns (attr)
Writes *count* bytes of data to a file beginning *offset* bytes from the beginning of the file. Returns the attributes of the file after the **write** takes place.

rename(dirfh, name, tofh, toname) returns (status)
Renames the file *name* in the directory *dirfh*, to *toname* in the directory *tofh*.

link(dirfh, name, tofh, toname) returns (status)
Creates the file *toname* in the directory *tofh*, which is a link to the file *name* in the directory *dirfh*.

symlink(dirfh, name, string) returns (status)
Creates a symbolic link *name* in the directory *dirfh* with value *string*. The server does not interpret the *string* argument in any way, just saves it and makes an association to the new symbolic link file.

readlink(fh) returns (string)
Returns the string which is associated with the symbolic link file.

mkdir(dirfh, name, attr) returns (fh, newattr)
Creates a new directory *name* in the directory *dirfh* and returns the new fhandle and attributes.

rmdir(dirfh, name) returns(status)
Removes the empty directory *name* from the parent directory *dirfh*.

readdir(dirfh, cookie, count) returns(entries)
Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, file id, and an opaque pointer to the next directory entry called a *cookie*. The *cookie* is used in subsequent **readdir** calls to start reading at a specific entry in the directory. A **readdir** call with the *cookie* of zero returns entries starting with the first entry in the directory.

statfs(fh) returns (fsstats)
Returns filesystem information such as block size, number of free blocks, etc.

New fhandles are returned by the **lookup**, **create**, and **mkdir** procedures which also take an fhandle as an argument. The first remote fhandle, for the root of a filesystem, is obtained by the client using the RPC based MOUNT protocol. The MOUNT protocol takes a directory pathname and returns an fhandle if the client has access permission to the filesystem which contains that directory. The reason for making this a separate protocol is that this makes it easier to plug in new filesystem access checking methods, and it separates out the operating system dependent aspects of the protocol. Note that the MOUNT protocol is the only place that UNIX pathnames are passed to the server. In other operating system implementations the MOUNT protocol can be replaced without having to change the NFS protocol.

The NFS protocol and RPC are built on top of the Sun External Data Representation (XDR) specification [3]. XDR defines the size, byte order and alignment of basic data types such as string, integer,.union, boolean and array. Complex structures can be built from the basic XDR data types. Using XDR not only makes protocols machine and language independent, it also makes them easy to define. The arguments and results of RPC procedures are defined using an XDR data definition language that looks a lot like C declarations. This data definition language can be used as input to an XDR protocol compiler which produces the structures and XDR translation procedures used to interpret RPC protocols [11].

Server Side

Because the NFS server is stateless, when servicing an NFS request it must commit any modified data to stable storage before returning results. The implication for UNIX based servers is that requests which modify the filesystem must flush all modified data to disk before returning from the call. For example, on a **write** request, not only the data block, but also any modified indirect blocks and the block containing the inode must be flushed if they have been modified.

Another modification to UNIX necessary for our server implimentation is the addition of a generation number in the inode, and a filesystem id in the superblock. These extra numbers make it possible for the server to use the inode number, inode generation number, and filesystem id together as the fhandle for a file. The inode generation number is necessary because the server may hand out an fhandle with an inode number of a file that is later removed and the inode reused. When the original fhandle comes back, the server must be able to tell that this inode number now refers to a different file. The generation number has to be incremented every time the inode is freed.

Client Side

The Sun implementation of the client side provides an interface to NFS which is transparent to applications. To make transparent access to remote files work we had to use a method of locating remote files that does not change the structure of path names. Some UNIX based remote file access methods use pathnames like *host:path* or */../host/path* to name remote files. This does not allow real transparent access since existing programs that parse pathnames have to be modified.

Rather than doing a "late binding" of file address, we decided to do the hostname lookup and file address binding once per filesystem by allowing the client to attach a remote filesystem to a directory with the *mount* command. This method has the advantage that the client only has to deal with hostnames once, at mount time. It also allows the server to limit access to filesystems by checking client credentials. The disadvantage is that remote files are not available to the client until a mount is done.

Transparent access to different types of filesystems mounted on a single machine is provided by a new filesystem interface in the kernel [13]. Each "filesystem type" supports two sets of operations: the Virtual Filesystem (VFS) interface defines the procedures that operate on the filesystem as a whole; and the Virtual Node (vnode) interface defines the procedures that operate on an individual file within that filesystem type. Figure 1 is a schematic diagram of the filesystem interface and how NFS uses it.

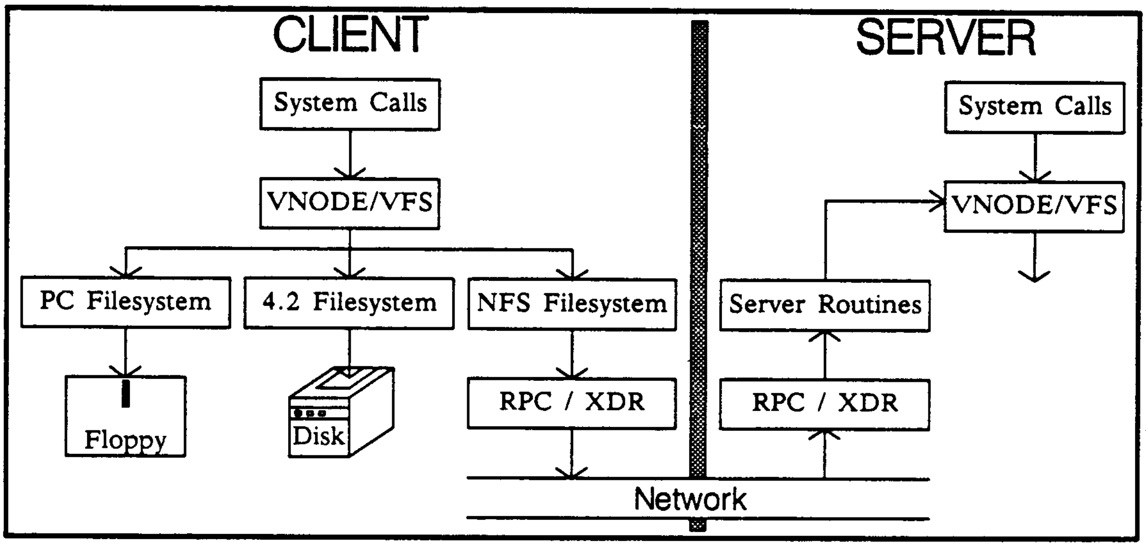


Figure 1

The Filesystem Interface

The VFS interface is implemented using a structure that contains the operations that can be done on a filesystem. Likewise, the vnode interface is a structure that contains the operations that can be done on a node (file or directory) within a filesystem. There is one VFS structure per mounted filesystem in the kernel and one vnode structure for each active node. Using this abstract data type implementation allows the kernel to treat all filesystems and nodes in the same way without knowing which underlying filesystem implementation it is using.

Each vnode contains a pointer to its parent VFS and a pointer to a mounted-on VFS. This means that any node in a filesystem tree can be a mount point for another filesystem. A **root** operation is provided in the VFS to return the root vnode of a mounted filesystem. This is used by the pathname traversal routines in the kernel to bridge mount points. The **root** operation is used instead of keeping a pointer so that the root vnode for each mounted filesystem can be released. The VFS of a mounted filesystem also contains a pointer back to the vnode on which it is mounted so that pathnames that include ".." can also be traversed across mount points.

In addition to the VFS and vnode operations, each filesystem type must provide **mount** and **mount_root** operations to mount normal and root filesystems. The operations defined for the filesystem interface are given below. In the arguments and results, vp is a pointer to a vnode, dvp is a pointer to a directory vnode and devvp is a pointer to a device vnode.

Filesystem Operations

mount(varies)	System call to mount filesystem
mount_root( )	Mount filesystem as root

VFS Operations

unmount(vfs)	Unmount filesystem
root(vfs) returns(vnode)	Return the vnode of the filesystem root
statfs(vfs) returns(statfsbuf)	Return filesystem statistics
sync(vfs)	Flush delayed write blocks

Vnode Operations

open(vp, flags)	Mark file open
close(vp, flags)	Mark file closed
rdwr(vp, uio, rwflag, flags)	Read or write a file
ioctl(vp, cmd, data, rwflag)	Do I/O control operation
select(vp, rwflag)	Do select
getattr(vp) returns(attr)	Return file attributes
setattr(vp, attr)	Set file attributes
access(vp, mode)	Check access permission
lookup(dvp, name) returns(vp)	Look up file name in a directory
create(dvp, name, attr, excl, mode) returns(vp)	Create a file
remove(dvp, name)	Remove a file name from a directory
link(vp, todvp, toname)	Link to a file
rename(dvp, name, todvp, toname)	Rename a file
mkdir(dvp, name, attr) returns(dvp)	Create a directory
rmdir(dvp, name)	Remove a directory
readdir(dvp) returns(entries)	Read directory entries
symlink(dvp, name, attr, toname)	Create a symbolic link
readlink(vp) returns(data)	Read the value of a symbolic link
fsync(vp)	Flush dirty blocks of a file
inactive(vp)	Mark vnode inactive and do clean up
bmap(vp, blk) returns(devp, mappedblk)	Map block number
strategy(bp)	Read and write filesystem blocks

bread(vp, blockno) returns(buf)	Read a block
brelse(vp, bp)	Release a block buffer

Notice that many of the vnode procedures map one-to-one with NFS protocol procedures, while other, UNIX dependent procedures such as **open**, **close**, and **ioctl** do not. The **bmap**, **strategy**, **bread**, and **brelse** procedures are used to do reading and writing using the buffer cache.

Pathname traversal is done in the kernel by breaking the path into directory components and doing a **lookup** call through the vnode for each component. At first glance it seems like a waste of time to pass only one component with each call instead of passing the whole path and receiving back a target vnode. The main reason for this is that any component of the path could be a mount point for another filesystem, and the mount information is kept above the vnode implementation level. In the NFS filesystem, passing whole pathnames would force the server to keep track of all of the mount points of its clients in order to determine where to break the pathname and this would violate server statelessness. The inefficiency of looking up one component at a time can be alleviated with a cache of directory vnodes.

Implementation

Implementation of NFS started in March 1984. The first step in the implementation was modification of the 4.2 kernel to include the filesystem interface. By June we had the first "vnode kernel" running. We did some benchmarks to test the amount of overhead added by the extra interface. It turned out that in most cases the difference was not measurable, and in the worst case the kernel had only slowed down by about 2%. Most of the work in adding the new interface was in finding and fixing all of the places in the kernel that used inodes directly, and code that contained implicit knowledge of inodes or disk layout.

Only a few of the filesystem routines in the kernel had to be completely rewritten to use vnodes. *Namei*, the routine that does pathname lookup, was changed to use the vnode **lookup** operation, and cleaned up so that it doesn't use global state. The *direnter* routine, which adds new directory entries (used by **create**, **rename**, etc.), was fixed because it depended on the global state from *namei*. *Direnter* was also modified to do directory locking during directory rename operations because inode locking is no longer available at this level, and vnodes are never locked.

To avoid having a fixed upper limit on the number of active vnode and VFS structures we added a memory allocator to the kernel so that these and other structures can be allocated and freed dynamically. The memory allocator is also used by the kernel RPC implementation.

A new system call, *getdirentries*, was added to read directory entries from different types of filesystems. The 4.2 *readdir* library routine was modified to use *getdirentries* so programs would not have to be rewritten. This change does, however, mean that programs that use *readdir* have to be relinked.

Beginning in March 1984, the user level RPC and XDR libraries were ported from the user-level library to the kernel, and we were able to make kernel to user and kernel to kernel RPC calls in June. We worked on RPC performance for about a month until the round trip time for a kernel to kernel **null** RPC call was 8.8 milliseconds on a Sun-2 (68010). The performance tuning included several speed ups to the UDP and IP code in the kernel.

Once RPC and the vnode kernel were in place the implementation of NFS was simply a matter of writing the XDR routines to do the NFS protocol, implementing an RPC server for the NFS procedures in the kernel, and implementing a filesystem interface which translates vnode operations into NFS remote procedure calls. The first NFS kernel was up and running in mid August. At this point we had to make some modifications to the vnode interface to allow the NFS server to do synchronous **write** operations. This was necessary since unwritten blocks in the server's buffer cache are part of the "client's state".

Our first implementation of the MOUNT protocol was built into the NFS protocol. It wasn't until later that we broke the MOUNT protocol into a separate, user level RPC service. The MOUNT server is a user level daemon that is started automatically by a mount request. It checks the file `/etc/exports` which contains a list of exported filesystems and the clients that can import them (see appendix 1). If the client has import permission, the mount daemon does a **getfh** system call to convert the pathname being imported into an fhandle which is returned to the client.

On the client side, the mount command was modified to take additional arguments including a filesystem type and options string. The filesystem type allows one *mount* command to mount any type of filesystem. The options string is used to pass optional flags to the different filesystem types at mount time. For example, NFS allows two flavors of mount, soft and hard. A hard mounted filesystem will retry NFS requests forever if the server goes down, while a soft mount gives up after a while and returns an error. The problem with soft mounts is that most UNIX programs are not very good about checking return status from system calls so you can get some strange behavior when servers go down. A hard mounted filesystem, on the other hand, will never fail due to a server crash; it may cause processes to hang for a while, but data will not be lost.

To allow automatic mounting at boot time and to keep track of currently mounted filesystems, the `/etc/fstab` and `/etc/mtab` file formats were changed to use a common ASCII format that is similar to the `/etc/fstab` format in Berkeley 4.2 with the addition of a type and an options field. The type field is used to specify filesystem type (nfs, 4.2, pc, etc.) and the options field is a comma separated list of option strings, such as rw, hard and nosuid (see appendix 1).

In addition to the MOUNT server, we have added NFS server daemons. These are user level processes that make an **nfsd** system call into the kernel, and never return. They provide a user context to the kernel NFS server which allows the server to sleep. Similarly, the block I/O daemon, on the client side, is a user level process that lives in the kernel and services asynchronous block I/O requests. Because RPC requests block, a user context is necessary to wait for read-ahead and write-behind requests to complete. These daemons provide a temporary solution to the problem of handling parallel, synchronous requests in the kernel. In the future we hope to use a light-weight process mechanism in the kernel to handle these requests [4].

We started using NFS at Sun in September 1984, and spent the next six months working on performance enhancements and administrative tools to make NFS easier to install and use. One of the advantages of NFS was immediately obvious; the *df* output below is from a diskless machine with access to more than a gigabyte of disk!

```
Filesystem             kbytes     used    avail capacity  Mounted on
/dev/nd0                 7445     5788      912    86%    /
/dev/ndp0                5691     2798     2323    55%    /pub
panic:/usr              27487    21398     3340    86%    /usr
fiat:/usr/src          345915   220122    91201    71%    /usr/src
panic:/usr/panic       148371   116505    17028    87%    /usr/panic
galaxy:/usr/galaxy       7429     5150     1536    77%    /usr/galaxy
mercury:/usr/mercury   301719   215179    56368    79%    /usr/mercury
opium:/usr/opium       327599    36392   258447    12%    /usr/opium
```

The Hard Issues

Several hard design issues were resolved during the development of NFS. One of the toughest was deciding how we wanted to use NFS. Lots of flexibility can lead to lots of confusion.

Filesystem Naming

Servers export whole filesystems, but clients can mount any sub-directory of a remote filesystem on top of a local filesystem, or on top of another remote filesystem. In fact, a remote filesystem can be mounted more than once, and can even be mounted on another copy of itself! This means that clients can have different "names" for filesystems by mounting them in different places.

To alleviate some of the confusion we use a set of basic mounted filesystems on each machine and then let users add other filesystems on top of that. Remember that this is policy, there is no mechanism in NFS to enforce this. User home directories are mounted on `/usr/servername`. This may seem like a violation of our goals because hostnames are now part of pathnames but in fact the directories could have been called `/usr/1`, `/usr/2`, etc.

Using server names is just a convenience. This scheme makes NFS clients look more like timesharing terminals because a user can log in to any machine and her home directory will be there. It also makes tilde expansion (where *~username* is expanded to the user's home directory) in the C shell work in a network with many machines.

To avoid the problems of loop detection and dynamic filesystem access checking, servers do not cross mount points on remote **lookup** requests. This means that in order to see the same filesystem layout as a server, a client has to remote mount each of the server's exported filesystems.

Credentials, Authentication and Security

NFS uses UNIX style permission checking on the server and client so that UNIX users see very little difference between remote and local files. RPC allows different authentication parameters to be "plugged-in" to the message header so we are able to make NFS use a UNIX flavor authenticator to pass uid, gid, and groups on each call. The server uses the authentication parameters to do permission checking as if the user making the call were doing the operation locally.

The problem with this authentication method is that the mapping from uid and gid to user must be the same on the server and client. This implies a flat uid, gid space over a whole local network. This is not acceptable in the long run and we are working on a network authentication method which allows users to "login" to the network [12]. This will provide a network-wide identity per user regardless of the user's identity on a particular machine. In the mean time, we have developed another RPC based service called the Yellow Pages (YP) to provide a simple, replicated database lookup service [5]. By letting YP handle `/etc/hosts`, `/etc/passwd` and `/etc/group` we make the flat uid space much easier to administer.

Another issue related to client authentication is super-user access to remote files. It is not clear that the super-user on a machine should have root access to files on a server machine through NFS. To solve this problem the server can map user *root* (uid 0) to user *nobody* (uid −2) before checking access permission. This solves the problem but, unfortunately, causes some strange behavior for users logged in as *root*, since *root* may have fewer access rights to a remote file than a normal user.

Concurrent Access and File Locking

NFS does not support remote file locking. We purposely did not include this as part of the protocol because we could not find a set of file locking facilities that everyone agrees is correct. Instead we have a separate, RPC based file locking facility. Because file locking is an inherently stateful service, the lock service depends on yet another RPC based service called the status monitor [6]. The status monitor keeps track of the state of the machines on a network so that the lock server can free the locked resources of a crashed machine. The status monitor is important to stateful services because it provides a common view of the state of the network.

Related to the problem of file locking is concurrent access to remote files by multiple clients. In the local filesystem, file modifications are locked at the inode level. This prevents two processes writing to the same file from intermixing data on a single write. Since the NFS server maintains no locks between requests, and a **write** may span several RPC requests, two clients writing to the same remote file may get intermixed data on long writes.

UNIX Open File Semantics

We tried very hard to make the NFS client obey UNIX filesystem semantics without modifying the server or the protocol. In some cases this was hard to do. For example, UNIX allows removal of open files. A process can open a file, then remove the directory entry for the file so that it has no name anywhere in the filesystem, and still read and write the file. This is a disgusting bit of UNIX trivia and at first we were just not going to support it, but it turns out that all of the programs that we didn't want to have to fix (*csh*, *sendmail*, etc.) use this for temporary files.

What we did to make open file removal work on remote files was check in the client VFS **remove** operation if the file is open, and if so **rename** it instead of removing it. This makes it (sort of) invisible to the client and still allows reading and writing. The client kernel then removes the new name when the vnode becomes inactive. We call this the 3/4 solution because if the client crashes between the **rename** and **remove** a garbage file is left on the server. An entry to *cron* can be added to clean up on the server, but, in practice, this has never been necessary.

Another problem associated with remote, open files is that access permission on the file can change while the file is open. In the local case the access permission is only checked when the file is opened, but in the remote case permission is checked on every NFS call. This means that if a client program opens a file, then changes the permission bits so that it no longer has read permission, a subsequent **read** request will fail. To get around this problem we save the client credentials in the file table at open time, and use them in later file access requests.

Not all of the UNIX open file semantics have been preserved because interactions between two clients using the same remote file cannot be controlled on a single client. For example, if one client opens a file and another client removes that file, the first client's **read** request will fail even though the file is still open.

Time Skew

Time skew between two clients or a client and a server can cause the times associated with a file to be inconsistent. For example, *ranlib* saves the current time in a library entry, and *ld* checks the modify time of the library against the time saved in the library. When *ranlib* is run on a remote file the modify time comes from the server while the current time that gets saved in the library comes from the client. If the server's time is far ahead of the client's it looks to *ld* like the library is out of date. There were only three programs that we found that were affected by this, *ranlib*, *ls* and *emacs*, so we fixed them.

Time skew is a potential problem for any program that compares system time to file modification time. We plan to fix this by limiting the time skew between machines with a time synchronization protocol.

Performance

The final hard issue is the one everyone is most interested in, performance.

Much of the development time of NFS has been spent in improving performance. Our goal was to make NFS comparable in speed to a small local disk The speed we were interested in is not raw throughput, but how long it takes to do normal work. To track our improvements we used a set of benchmarks that include a small C compile, tbl, nroff, large compile, f77 compile, bubble sort, matrix inversion, make, and pipeline.

To improve the performance of NFS, we implemented the usual read-ahead and write-behind buffer caches on both the client and server sides. We also added caches on the client side for file attributes and directory names. To increase the speed of read and write requests, we increased the maximum size of UDP packets from 2048 bytes to 9000 bytes. We cut down the number of times data is copied by implementing a new XDR type that does XDR translation directly into and out of *mbufs* in the kernel.

With these improvements, a diskless Sun-3 (68020 at 16.67 Mhz.) using a Sun-3 server with a Fujitsu Eagle disk, runs the benchmarks faster than the same Sun-3 with a local Fujitsu 2243AS 84 Mega-byte disk on a SCSI interface.

The two remaining problem areas are **getattr** and **write**. The reason is that *stat*-ing files causes one RPC call to the server for each file. In the local case the inodes for a whole directory end up in the buffer cache and then *stat* is just a memory reference. The **write** operation is slow because it is synchronous on the server. Fortunately, the number of **write** calls in normal use is very small (about 5% of all calls to the server, see appendix 2) so it is not noticeable unless the client writes a large remote file.

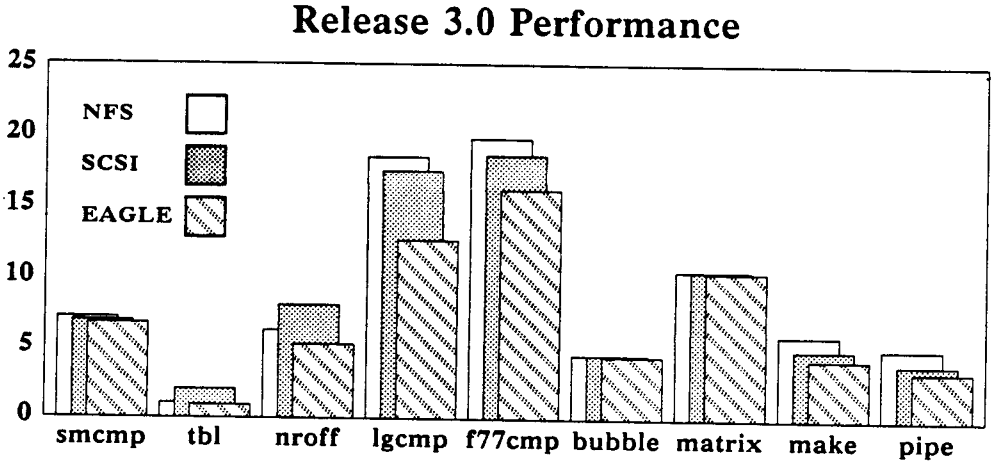


Figure 3

In Figure 3, above, we show some benchmark results comparing NFS and local SCSI disk performance for the current Sun software release. The scale on the left is unitless numbers. It is provided to make comparison easier.

Since many people base performance estimates on raw transfer speed we also measured those. The current numbers on raw transfer speed are: 250 kilobytes/second for read (cp bigfile /dev/null) and 60 kilobytes/second for write on a Sun-3 with a Sun-3 server.

Other Remote Filesystems

Why, you may ask, do we need NFS when we already have Locus[14], Newcastle Connection[15], RFS[8], IBIS[16] and EFS[10]. In most cases the answer is simple: NFS is designed to handle non-homogeneous machines and operating systems, it is fast, and you can get it today. Other than the Locus system, which provides file replication and crash recovery, the other remote filesystems are very similar to each other.

RFS vs NFS

The AT&T Remote Filesystem (RFS), which has been demonstrated at USENIX and UniForum conferences but not yet released, will provide much of the same functionality as NFS. It allows clients to mount filesystems from a remote server and access those files in a transparent way. The differences between them mostly stem from the basic design philosophies. NFS provides a general network service, while RFS provides a distributed UNIX filesystem[9]. This difference in philosophy shows up in many different areas of the designs.

Networking

RFS does not use standard network transport protocols, like UDP/IP. Instead it uses a special purpose transport protocol which has not been published, and implementations of it are not generally available. This protocol cannot easily be replaced because RFS depends on properties of the transport virtual circuit to determine when a machine has crashed. NFS uses the RPC layer to hide the underlying protocols, which makes it easy to support different transport protocols without having to change the NFS protocols.

RFS does not use a remote procedure call mechanism, instead it extends the semantics of UNIX system calls so that a system call which accesses a remote file goes over the network and continues execution on the server. When the system call is finished, the results are returned to the client. This protocol is complicated by the fact that both client and server can interrupt a remote system call. In addition, the system calls which deal with filenames had to be modified to handle a partial lookup on the server when a client mount point is encountered in the pathname. In this case the server looks up part of the name then returns control to the client to look up the rest.

Non-Homogeneous Machines and Operating Systems

While NFS currently runs on 16 different vendors hardware, and under Berkeley 4.2, Sun OS, DEC Ultrix, System V.2, VMS and MS/DOS, RFS will run only System V.3 based UNIX systems. The NFS design is based on the assumption that most installations have many different types of machines on their network, and that these machines run widely varying systems. The RFS protocol includes a canonical format for data to help support different machine architectures, but no attempt is made to support operating systems other than System V.3. The NFS design does not try to predict the future. Instead, it includes enough flexibility to support evolving software, hardware, and protocols.

Flexibility

Because RFS is built on proprietary protocols with UNIX semantics built in, it is hard to imagine using those protocols from different operating systems. NFS, on the other hand, provides flexibility through the RPC layer. RPC allows different transport protocols, authentication methods, and server versions to be supported in a single implementation. This allows us, for example, to use an encrypted authentication method for maximum security among workstations, while still allowing access by PC's using a simpler authentication method. It also makes protocol evolution easier since clients and servers can support different versions of the RPC based protocols simultaneously.

RFS uses streams [7] to hide the details of underlying protocols. This should make it easy to plug in new transport protocols. Unfortunately, RFS uses the virtual circuit connection of the transport protocol to detect server and client crashes*. This means that even the reliable byte stream protocol TCP/IP cannot be plugged in because TCP connections do not go away when one end crashes unless there is data flowing at the time of the crash.

Crash Recovery

The RFS uses a stateful protocol. The server must maintain information about the current mount points of all of its clients, the open files, directories, and devices held by its clients, as well as the state of all client requests that are in progress. Because it would be very difficult and costly for the client to rebuild the server's state after a server crash, RFS does not do server crash recovery. A server or client crash is detected when the protocol connection fails, at which point all operations in progress to that machine are aborted. When an RFS server crashes it is roughly equivalent, from the client's point of view, to losing a local disk.

If server crashes are rare events doing no recovery is acceptable, however, keep in mind that network delays, breaks, or overloading usually cannot be distinguished from a machine crash. As networks grow the possibility of network failures increases, and as the connectivity of the network increases so does the chance of a client or server crash. We decided early in the design process that NFS must recover gracefully from machine and network problems. NFS does not need to do crash recovery on the server because the server maintains no state about its clients. Similarly, the client recovers from a server crash simply by resending a request.

Administration

There are two major differences between administration of NFS and RFS. The use of a uid mapping table on RFS servers removes the need for uniform uid to user mapping through out the network. NFS assumes a uniform uid space and we provide the Yellow Pages service to make distribution and central administration of system databases (like /etc/passwd and /etc/group) easier. NFS also has a MOUNT RPC service for each machine acting as a server. The exported filesystem information is maintained on each machine and made available by this service. RFS uses a centralized name service running on one machine on the network to keep track of advertised filesystems for all servers. A centralized name service was not acceptable in NFS because it allows a single point of failure for the whole network, and it forces all clients and servers to use the same protocol for exchanging mount information. By having a separate protocol for the MOUNT service we can support different filesystem access checking and different operating system dependent features of the mount operation.

* The exclusive use of transport properties to drive session semantics is a common design flaw in many network applications.

UNIX Semantics

NFS does not support all of the semantics of UNIX filesystems on the client. Removing an open file, append mode writes, and file locking are not fully implemented by NFS. RFS does implement 100% of the UNIX filesystem semantics. However, if a server crashes or a filesystem is taken out of service, client applications can see error conditions which normally could only happen due to a disk failure. Since this is an error condition that is so severe that it usually means that the whole system has failed, most applications will not even try to recover.

Availability

NFS has been a product for more than a year. Source and support for NFS on Berkeley 4.2 BSD is available through Sun and Mt. Xinu, and for System V.2 through Lachman Associates, The Instruction Set, and Unisoft. RFS has not yet been released.

Conclusion

For a small network of machines all running System V.3, RFS is the obvious choice for remote access to files since it will come with V.3 and it implements all of the UNIX semantics. For a large network or a network of mixed protocols, machine types, and operating systems, NFS is the better choice. It should be understood that NFS and RFS are not mutually exclusive. It will be possible to run both on a single machine.

Porting Experience

In the many ports of NFS to foreign hardware and systems we have found only a few places where additions to the protocol would be helpful. The IBM PC client side port was done almost exclusively from the protocol specification, and a simple, user-level server was also implemented from the specification.

NFS has been ported to five different operating systems, two of which are not UNIX based, and to many different types of machines. Each port had its own interesting problems.

The first port of NFS was to a VAX 750 running Berkeley 4.2 BSD. This was also the easiest port since our code is based on 4.2 UNIX. Modifying the kernel to use the vnode/VFS interface was the most time consuming part of the porting effort. Once the vnode/VFS interface was in, the NFS and RPC code pretty much just dropped in. Some libraries had to be updated, and programs that read directories had to be recompiled. The whole port took about two man-weeks to complete. This port was then used as the distribution source for later ports.

The System V.2 port was done in a joint effort by Lachman Associates and The Instruction Set on a VAX 750. In order to avoid having to port the Berkeley networking code to the System V kernel an Exelan board was used. The Exelan board handles the ethernet, IP, and UDP layers. A new RPC transport layer had to be implemented to interface to the Exelan board. Adding the vnode/VFS interface to the System V kernel was the hardest part of the port.

The port to the IBM PC, done by Geoff Arnold and Kim Kinnear at Sun, was complicated by the need to add a "redirector" layer to MS/DOS to catch system calls and redirect them. An implementation of UDP/IP also had to be added before RPC could be ported. The NFS client side implementation is written in assembler and occupies about 40K bytes of space. Currently, remote *read* operations are faster than a local hard disk access but remote *write* operations are slower. Over all, performance is about the same for remote and local access.

DEC has ported NFS to Ultrix on a Microvax II. This port was harder than the 4.2 port because the Ultrix release that was used is based on Berkeley 4.3beta. The most time consuming part of the port was, again, installing the vnode/VFS interface. This was complicated by the fact that Berkeley has made many changes to much of the kernel code that deals with inodes.

Another interesting port, while not a different operating system, was the Data General MV 4000 port. The DG machine runs System V.2 with Berkeley 4.2 networking and filesystem added. This made the RPC and vnode/VFS part of the port easy. The hard part was XDR. The MV 4000 has a word addressed architecture, and character pointers are handled very differently than word pointers. There were many places in the code, and especially in the XDR routines that assumed that (char *) == (int *).

As an aid to porting we have implemented a user-level version of the NFS server (UNFS). It uses the standard RPC and XDR libraries and makes system calls to handle remote procedure call requests. The UNFS can be ported to non-UNIX operating systems by changing the system calls and library routines that are used. Our benchmarks show it to be about 80% of the performance of a kernel based NFS server for a single client and server.

The VMS implementation is for the server side only. The basic port was done by Dave Kashten at SRI. He started with the user-level NFS server and used the EUNICE UNIX-emulation libraries to handle the UNIX system calls. The RPC layer was ported to use a version of the Berkeley networking code that runs under VMS. Some caching was added to the libraries to speed up the system call emulation and to perform the mapping from UNIX permission checking to VMS permission checking.

At the UniForum conference in February 1986, all of the completed NFS ports were demonstrated. There were 16 different vendors and five different operating systems all sharing files over an ethernet.

Also at UniForum; IBM officially announced their RISC based workstation product, the RT. Before the announcement, NFS had already been ported to the RT under Berkeley 4.2 BSD by Mike Braca at Brown University.

Conclusions

We think that the NFS protocols, along with RPC and XDR, provide the most flexible method of remote file access available today. To encourage others to use NFS, Sun has made public all of the protocols associated with NFS. In addition, we have published the source code for the user level implementation of the RPC and XDR libraries.

Acknowledgements

There were many people throughout Sun who were involved in the NFS development effort. Bob Lyon led the NFS group and helped with protocol issues, Steve Kleiman implemented the filesystem interface in the kernel from Bill Joy's original design, Russel Sandberg ported RPC to the kernel and implemented the NFS virtual filesystem, Tom Lyon designed the protocol and provided far sighted inputs into the overall design, David Goldberg worked on many user level programs, Paul Weiss implemented the Yellow Pages, and Dan Walsh is the one to thank for the performance of NFS. The NFS consulting group, headed by Steve Isaac, has done an amazing job of getting NFS out to the world.

References

[1] B. Lyon, "Sun Remote Procedure Call Specification," Sun Microsystems, Inc. Technical Report, (1984).

[2] R. Sandberg, "Sun Network Filesystem Protocol Specification," Sun Microsystems, Inc. Technical Report, (1985).

[3] B. Lyon, "Sun External Data Representation Specification," Sun Microsystems, Inc. Technical Report, (1984).

[4] J. Kepecs, "Lightweight Processes for UNIX Implementation and Applications," USENIX (1985).

[5] P. Weiss, "Yellow Pages Protocol Specification," Sun Microsystems, Inc. Technical Report, (1985).

[6] J. M. Chang, "SunNet," USENIX (1985).

[7] D.L. Presotto and D. M. Ritchie, "Interprocess Communication in the Eighth Edition UNIX System," USENIX Conference Proceedings, (June 1985).

[8] P. J. Weinberger, "The Version 8 Network File System," USENIX Conference Proceedings, (June 1985).

[9] M. J. Hatch, et al., "AT&T's RFS and Sun's NFS, A Comparison of Heterogeneous Distributed File Systems," UNIX World, (December 1985).

[10] C. T. Cole, et al., "An Implementation of an Extended File System for UNIX," USENIX Conference Proceedings, (June 1985).

[11] B. Taylor, "A protocol compiler for RPC," Sun Microsystems, Inc. Technical Report, (December 1985).

[12] B. Taylor, "A Secure Network Authentication Method for RPC," Sun Microsystems, Inc. Technical Report, (November 1985).

[13] S. R. Kleiman, "An Architecture for Multiple File Systems in Sun UNIX," Sun Microsystems, Inc. Technical Report, (October 1985).

[14] Popek, et al., "The LOCUS Distributed Operating System," Operating Systems Review ACM, (October 1983).

[15] D. R. Brownbridge, et al., "The Newcastle Connection or UNIXes of the World Unite!," Software -- Practice and Experience, (1982).

[16] W. F. Tichy, et al., "Towards a Distributed File System," USENIX Conference Proceedings, (June 1985).

Appendix 1

/etc/fstab and /etc/mtab format

The format of the filesystem database files /etc/fstab and /etc/mtab were changed to include type and options fields. The type field specifies which filesystem type this line refers to, and the options field specifies mount and run time options. The options field is a list of comma separated strings. This allows new options to be added, for example when a new filesystem type is created, without having to change the library routines that parse these files. The example below is the /etc/fstab file from a diskless machine.

```
(Filesystem                  mount point       type options)

/dev/nd0                     /                 4.2  rw            1 1
/dev/ndp0                    /pub              4.2  ro            0 0
speed:/usr.MC68010           /usr              nfs  ro,hard       0 0
#opium:/usr/opium            /usr/opium        nfs  rw,hard       0 0
speed:/usr.MC68020/speed     /usr/speed        nfs  rw,hard       0 0
panic:/usr/src               /usr/src          nfs  rw,soft,bg    0 0
titan:/usr/doctools          /usr/doctools     nfs  ro,soft,bg    0 0
panic:/usr/panic             /usr/panic        nfs  rw,soft,bg    0 0
panic:/usr/games             /usr/games        nfs  ro,soft,bg    0 0
wizard:/arch/4.3alpha        /arch/4.3         nfs  ro,soft,bg    0 0
sun:/usr/spool/news          /usr/spool/news   nfs  ro,soft,bg    0 0
krypton:/usr/release         /usr/release      nfs  ro,soft,bg    0 0
crayon:/usr/man              /usr/man          nfs  soft,bg       0 0
crayon:/usr/local            /usr/local        nfs  ro,soft,bg    0 0
topaz:/MC68010/db/release    /usr/db           nfs  ro,soft,bg    0 0
eureka:/usr/ileaf            /usr/ops          nfs  soft,bg       0 0
wells:/pe                    /pe               nfs  rsize=1024    0 0
```

Mount Access Permission: the /etc/exports File

The file /etc/exports is used by the server's MOUNT protocol daemon to check client access to filesystems. The format of the file is <filesystem> <access-list>. If the access list is empty the filesystem is exported to everyone. The access-list consists of machine names and netgroups. Netgroups are like mail aliases, a single name refers to a group of machines. The netgroups database is accessed through the Yellow Pages. Below is and example /etc/exports file from a server.

```
(filesystem         access-list)

/usr                argon krypton
/usr/release
/usr/misc
/usr/local
/usr/krypton        argon krypton phoenix sundae
/usr/3.0/usr/src    systems
/usr/src/pe         pe-users
```

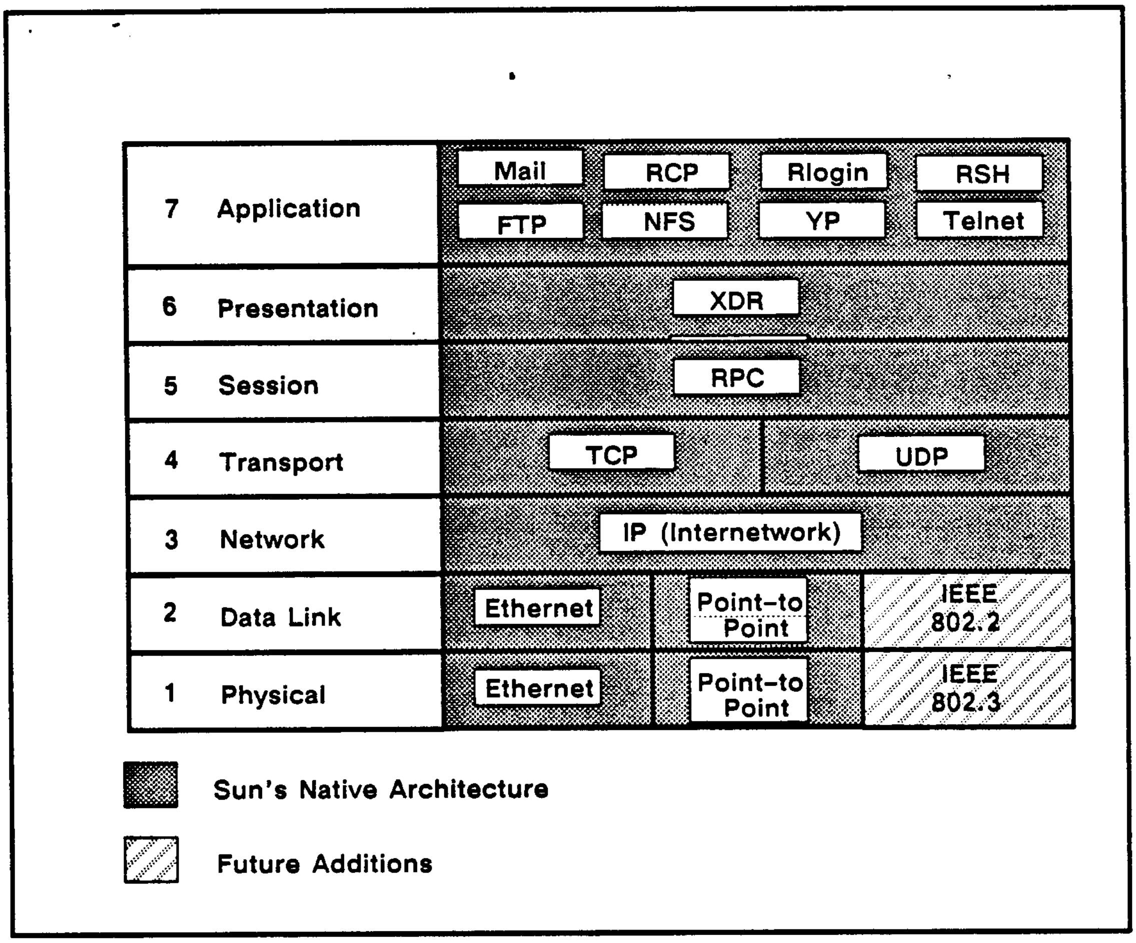
Appendix 2

Below are the server NFS and RPC statistics collected from a typical server at Sun. Statistics are collected automatically each night, using the *nfsstat* command, and sent to a list of system administrators. The statistics are useful for load balancing and detecting network problems. Note that 1499689 calls/day = 62487 calls/hour = 17 calls/second, average over twenty four hours for one server!

```
Server rpc:
calls       badcalls    nullrecv    badlen      xdrcall
1499688     0           0           0           0

Server nfs:
calls       badcalls
1499688     0

null        getattr     setattr     root        lookup      readlink
0  0%       79897   5%  708   0%    0  0%       760709 50%  116712   7%

read        wrcache     write       create      remove      rename
452090 30%  0  0%       50151   3%  25394   1%  5605   0%   687  0%

link        symlink     mkdir       rmdir       readdir     fsstat
683   0%    83   0%     1  0%       1  0%       6960   0%   7  0%
```

Appendix 3

Sun Protocols in the ISO Open Systems Interconnect Model



7	Application	Mail	RCP	Rlogin	RSH			
		FTP	NFS	YP	Telnet			
6	Presentation	XDR						
5	Session	RPC						
4	Transport	TCP	UDP					
3	Network	IP (Internetwork)						
2	Data Link	Ethernet	Point-to Point	IEEE 802.2				
1	Physical	Ethernet	Point-to Point	IEEE 802.3				

▩ Sun's Native Architecture

▨ Future Additions

Tools for the Maintenance and Installation of a Large Software Distribution

D.M. Tilbrook
P.R.H. Place

Imperial Software Technology

ABSTRACT

This paper describes the problems inherent in developing and maintaining a large software distribution. A strategy for software development and its relevance to these problems is discussed. Brief outlines of policies that implement the strategy are then presented. The IST implementation is described with examples of the support tools developed. Finally, more detailed descriptions of some tools are presented, with particular reference to *pmak*(1), a front end to *make*(1)[1].

1. INTRODUCTION

In general, programming techniques applicable to small scale software projects do not scale up to large, or even medium scale, projects. At Imperial Software Technology (IST), we are developing software (approximately 3,000 files, 300K lines, 7.2M bytes of source code) in two projects: ISTAR, a project developing an integrated project support environment, and 7001, a project researching into software development, described in this paper. The ISTAR project uses much of the software produced by 7001, both for development and delivery. The continuing development, application and distribution of the 7001 software has highlighted some of the problems inherent in software development. In an attempt to resolve these problems, an overall software strategy has been evolved and adopted. Further, particular policies for carrying out this strategy have been developed, both in the form of standard practices for the development of software, and as tools to assist that development.

This paper discusses the software problems and describes how our software strategy resolves these problems. The manner in which this strategy is implemented at IST is described and some of the tools will be discussed in detail. Particular attention is paid to tools that are part of the software construction process. The paper contains the following sections:

- Problems
- Software Strategy
- Policies
- Pmak
- Use at IST
- Conclusions

The work described in this paper developed on and for UNIX[2] systems. As such it refers to UNIX tools and facilities. It is expected that the reader has some

[1] The term "make" will be used extensively throughout this paper as a verb, a proper noun and an adjective. In each case, its use should be clear from the context. Therefore most occurrences will not be distinguished in any way.

knowledge of the concepts and use of *make*(1) and some of the other standard UNIX tools.

2. *PROBLEMS*

The development and maintenance of 7001 software has enabled us to identify a number of problems that occur in large software projects. In this section, we describe these problems and also give an indication as to why *make*(1), as used in the past is inadequate for large scale projects.

The problems that we describe are:

- Education.
- Interdependency of Construction.
- Application of the Software.
- Identification.
- System Variations.
- Problems with Make
  - Differences Between Versions
  - Lack of Standards
  - Overloading
  - Duplication of Information
  - Weakness of the Time-Dependency Relation
  - The Remake Problem
  - Difficulty of Make Script Maintenance
  - Dynamic Dependency
  - Include Facility
  - No Conditionals
  - No Local Assignments

2.1 Education

Staff joining a company will have a wide range of programming practices (depending on programming languages used and previous experience). It is important for a company to educate these staff in the "company style" (this should include preferred coding standards and available tools). Unfortunately, when projects have to deliver products in short time scales, this education is often thought to be a luxury that may be omitted. This leads to a number of people developing code for a client without attempting to standardise their coding practices. The resulting diversity of style means that it is hard to adequately assure the quality of the deliverable products.

2.2 Interdependency of Construction

Software often contains complex dependencies, where the construction of a program is dependent upon the prior construction (and installation) of other programs or libraries. It is not an easy task to specify in a clear and precise way the order in which libraries and programs should be constructed.

A particular example of this is the mail system, *ma*(1), used at IST (a system which rivals the Rand mail handler in complexity - but not in code size). The construction of *ma*(1) depends upon the prior installation of *arlo*(1) which in its

[2] UNIX is a trademark of AT&T Bell Laboratories.

turn depends upon TIPs, TIPs requires *xdb*(1) which is dependent upon the IST library.

2.3 Application of the Software

We use the 7001 software in three different ways: As a research tool, as a production system and as part of a product. This entails the maintenance of three versions of the software: the research version which is updated on a continuous basis; the production version, used within IST and updated every two or three months; and the distribution version, delivered to customers and updated at longer intervals. This creates the problem of maintaining consistent information over the different versions of the software (a problem that is only partially solved by considering the research version as the master copy).

We also create special test versions of the software for particular purposes. For example, we recently created another copy of the software to enable the porting of ISTAR to a new machine (and operating system).

2.4 Identification

With a number of people changing the source code, there is little stability in a system and a piece of software can suddenly stop working because a change to some other piece of software has been made. We have to be able to keep track of these changes in order to determine responsibility for the code and hence to maintain control over the development of the project.

2.5 System Variations

At IST, we use a number of different machines running different versions of UNIX. In particular, we have VAX/750's[3] and a 3B2 running System VR2, as well as a number of 68000's running Uniplus System V, a Pyramid 90 running OSx and a VAX/750 running 4.3bsd.[4]

With this variation in machines and operating systems, we have had to face the problem of "standard" tools varying between machines, (e.g., *install*(1) and *lint*(1)), programs being installed in different directories (e.g., *lint*(1)), header files being in different locations (e.g., *wait.h*), files having different names (e.g., *string.h* vs. *strings.h* vs. nothing at all), routines having different names (e.g., *strchr*(3) vs. *index*(3)), and routines returning values of different types (e.g., *sprintf*(3)). One previous solution to this problem has been to overload make, however, this often results in very complicated make scripts.

2.6 Problems with Make

Whilst the problems described above are significant, the most difficult problems faced are those caused by make. Make is probably the most important tool in an installer's tool-kit and there can be little doubt that without it, or some tool with a similar function and power, the installation of a large software system would be almost impossible.

However, make does have a number of short-comings.

2.6.1 Differences Between Versions There are at least three versions of make in use. For the most part the fundamental syntax and function are the same. However, there are substantial differences with respect to the handling of the construction rules and archive files.

[3] VAX is a trademark of Digital Equipment Corporation

[4] We have also run the 7001 software on 4.1bsd, 4.2bsd, Ultrix, System III and V8.

Since some versions of make are superior to others, sites often use a "non-native" version. This makes it difficult to prepare complicated make scripts for a particular system since the version of make in use at the target site may not be the standard version for the system.

2.6.2 Lack of Standards Despite the use of make for about ten years, no dominant style of use has evolved. There are a number of target names that are used frequently in make scripts (e.g., all, clobber and clean) but the associated functions vary from script to script. Comprehensive make scripts tend to be exceedingly large and complicated. Yet it is almost essential to read a make script before use.

2.6.3 Overloading A major cause of complexity in make scripts is the tendency for programmers to overload the script with rules and operations not concerned with construction. It is quite common to find commands to lint, print, vgrind, delta or get sccs s-files, build ctags files and even invoke editors on source files in a script, though none of these operations are part of the fundamental purpose of a make script which is to construct and install programs and their related data files. Make does provide convenient mechanisms to perform and select these other operations, but their inclusion complicates the scripts and hides essential information.

2.6.4 Duplication of Information Make scripts often have information duplicated (in slightly different forms) in different parts of the script. For example, a single file name may appear, in various forms (e.g., *echo.c*, *echo.o*, *echo*, */bin/echo*) throughout the script. File names often appear in lists (where each file is to be treated in the same way) and the ordering of these lists may be significant, however, it is not possible to determine the significance of the lists without examining the entire make script.

A further cause of duplication of information is that the current form of make forces a conscientious software designer to give the name of an object library file at least twice. Consider a make script to compile a program that depends on a library, then the library name should be given in the dependency list for the program and also in the compilation command.

This requirement for the multiple entry of information leads to either incomplete, inconsistent or redundant specifications. For 'example, some programmers create a variable that names all the libraries used by the programs constructed in the make script and use the variable whenever libraries names are required (e.g., the dependency lists and compilation commands). This can lead to confusion and errors.

2.6.5 Weakness of the Time-Dependency Relation One cause of information duplication is that the time-dependency relation is difficult to limit, non-transitive and inadequate.

For example, assume that */bin/echo* depends on a local version of echo which in turn depends on the source file *echo.c*. Even though */bin/echo* is more recent than the source file *echo.c*, if make is directed to construct */bin/echo* and the local version of *echo* does not exist, then both *echo* and */bin/echo* will be unnecessarily re-constructed.

To illustrate the lack of transitivity, assume that echo.c "#include"s a file *stdio.h*. The construction of the file *echo.c* does not depend on *stdio.h*, but *echo* and *echo.o*'s constructions do depend on *stdio.h*. The inability of make to express such a dependency means that all such transitive dependencies must be unfolded and stated explicitly, thus greatly increasing the size, complexity and construction time of make scripts.

A further problem with time-dependency is the propagation of redundant operations due to changes in a file's modification time but not its contents. For example, assume that *lex.l* is a *lex*(1) script that includes a header file generated by *yacc*(1)'s processing of the file *yacc.y*. Every time *yacc.y* is changed, *yacc*(1) is rerun which will re-produce the header file *y.tab.h*. This should in turn force the re-compilation of *lex.c* (a process that is very expensive). To avoid this unnecessary step, a copy of *y.tab.h*, say *y.tab.th*, is kept with *lex.o* depending on *y.tab.th* (*y.tab.th* only being updated when it differs from *y.tab.h*). Let *y.tab.h* be re-produced, but without changing its contents. Then, on each subsequent invocation of the make script, *y.tab.h* will be compared with *y.tab.th*.

2.6.6 The Remake Problem Another problem with the time dependency relation is that the only way to force the reconstruction of a target is to remove it or to *touch*(1) a file on which it depends.

This is a particular problem when dealing with programs that generate object or fixed format files. For example, one of the TIPs programs, *tmkprof*(1), converts a textual description of a data base into an encoded representation (the profile). Any change in this encoding, naturally requires that all existing profiles be recreated. It is not desirable to have the profiles depend on *tmkprof*(1) as the slightest change (e.g., fixing a spelling mistake in an error message) would force the regeneration of all profiles, which might, in turn, cause many other unnecessary and expensive regenerations. But there is no convenient or commonly used mechanism within make to handle such a problem.

2.6.7 Difficulty of Make Script Maintenance Because, as a consequence of the above problems, make scripts are often very complicated, changing them is a difficult and error-prone task. Due to time constraints or lack of concern, make scripts tend to be "hacked" and not designed. Additions are made by duplicating lines that are "close" to what is required and then making necessary changes to have the script work correctly. This leads to unstructured, over-complicated and incorrect scripts.

2.6.8 Dynamic Dependency A product may depend upon some dynamic list of data files. An obvious dependency to use is:

```
target: *.d
```

However, although make will correctly rebuild *target* if a new data file is added, it will fail to rebuild *target* if one of the data files is removed.

2.6.9 Include Facility There is no universal or useful include facility. The standard make on bsd distributions does not support an include facility. The AT&T versions have a form of include that is rendered inadequate by requiring the include parameter to be an absolute pathname (i.e., there is no search path and variables may not be used to resolve the location of the desired file).

For portability, it is essential that make scripts be able to include files to establish environmental settings and have a search path that can be externally specified.

2.6.10 No Conditionals The lack of conditional tests within make means that it is not easy to dynamically select (or suppress) a given construction. Due to syntactic problems, using the shell to perform tests is both a frustrating and difficult exercise. It is also a non-trivial task to ensure that errors are handled correctly. For example, one has to ignore the result of any test that might legitimately return a non-zero status, but this means that real errors aren't handled properly.

2.6.11 No Local Assignments It would be desirable to use make's limited variable facility to be able to design and use prototype constructions. Unfortunately, because make assigns values to its variables in the first pass and uses the values in the second pass, only the last assigned value will be used.

2.7 Disclaimer

Before leaving this section, we must reiterate that, despite the problems we have identified, make is the single most important tool in our approach to installing and porting software.[5] Most of our efforts have been directed at the development of techniques and tools to use make more effectively, not to replace it. In fact, it is now our opinion that not only should make not be enhanced, but should be reduced in power in order to achieve improved performance and to eliminate some features that we don't need and interfere with its more effective use, which is to say, make make make things well.

3. *SOFTWARE STRATEGY*

In order to solve the problems described in the previous section in an effective manner, a software strategy has been evolved. The main aim of the strategy was to reduce the difficulty of developing and maintaining code. A second important aim was to make it easier to follow, than to ignore the strategy. The following aspects of the software strategy are described.

- Isolation.
- Provision of Tools.
- Single Sourcing.
- Disciplined Organisation of Source Code.
- Locatability.
- Self Documenting Programs.
- Reducing the Complexity of Makefiles.
- Auditing.

3.1 *Isolation*

A primary assumption is that the software is to be installed on a "vanilla" UNIX system, which is to say that none of the bugs have been fixed and no enhancements are available. Thus, the software installation process must make minimal assumptions about its environment. Another consideration is that the software should be totally isolated from the rest of the system. It should be possible to install the software, test it and remove it without having affected the original system. It should also be possible to ensure that such a test does not use previously installed or enhanced parts of the environment.

3.2 *Provision of Tools*

As has been stated, many of the tools available under UNIX vary from system to system or have bugs which make them less than helpful. Where necessary, distributions must include tools to rectify known bugs or to provide missing facilities.

For example, we provide versions of *basename*(1) and *ctime*(3) which are used by the 7001 software instead of the standard versions. In keeping with the isolation strategy, the 7001 replacements are not installed in the standard libraries or directories.

3.3 *Single Sourcing*

Another important part of the strategy is single sourcing of information. If information is expressed in only one location, then there is a much greater chance of that information being in step with the system. Therefore, all the site dependent information should be contained in a single location and processed to create the

[5] It might bear mentioning that Stu Feldman, the original creator of make, will soon be dt's manager (as much as is possible).

relevant files. Where possible, information about constructions or installations should be stored in a single well known location.

For example, we use a file called *system.h* to contain the version of UNIX being used. All other files that refer to the version (or name) of the system use this file. A further example are the TIPs databases of library functions. They are used to generate *lint*(1) libraries, to create *xdb*(1) databases (used to provide online glossaries and references), the software inventory database entries, as well as UPM sections. As a logical extension of this strategy and these databases, we have experimented both with extracting C source from the database entry and embedding the database entry in the C source. The former makes writing C code harder and the latter requires a small change to the C pre-processor. Other examples of single sourcing are presented later in the paper.

3.4 Disciplined Organisation of Source Code

A further feature of the strategy is to separate files by their use (or function) into different directories. This leads to a software organisation where only one library is constructed from each directory and where shared header files are placed in separate directories.

For example, we avoid files with the name **main.c**, the reason being that main.c does not help identify the function of the program, whereas *passwd.c* suggests that the file is the main source file for *passwd*(1).[6]

3.5 Locatability

Another aim is to be able to easily locate the source for any program. Therefore, we must construct a source inventory database as well as tools to support its construction and interpretation.

We could use *find*(1), but on V7 and System V it is unacceptably slow. Even the improved version, on bsd systems, provides a great deal of extraneous information (e.g., "find printf" will find all occurrences of printf.c, but also printf.o and any other files with printf in their name such as wwprintf.c).

3.6 Self Documenting Programs

In response to the problem of training new personnel, programs should be substantially self-documenting. Since manual entries often provide too much information, at the wrong level, in the wrong order and, perhaps, out of date, help information should be generated from, and incorporated into, each program.

3.7 Reducing the Complexity of Makefiles

In order to resolve some of the problems with make, we should reduce the complexity of make files, or even render them unnecessary. Tools must be provided to support the creation and maintenance of make files and to provide alternatives to make for non-construction operations.

As an example, the existence of a source file in a directory should be sufficient indication that the file is to be used. Information about a program's construction should be stored in the program source, not in the make script.

3.8 Auditing

In order to solve the problem of traceability, source code control techniques should be used as a default action, rather than as an afterthought. Each time a file is

[6] There are 53 files called main.c in our 4.3bsd source trees. Yet there is no adb.c, make.c, lex.c, yacc.c and the only sh.c is source for the program installed as csh.

changed (or installed), a record should be kept of the reasons for the change, the person responsible for the change and the change itself.

SCCS or a similar system, is necessary to perform this record keeping. However, we also require that a record of any changes be kept in a form that may be used by other tools. In order that this record accurately reflects changes, the record keeping should be automatic, requiring no effort from the user. Further, each time a program is installed, a record should be kept of the installation, so that all the changes to a system are recorded. Again, this record should be automatic. Tools are then needed in order to create and manipulate these records.

4. POLICIES

Over the last decade, subsets of the current 7001 software have been installed on a variety of machines including V6, V7, V8, PWB, 4.1bsd and System III. In general, these installations were prompted by changes of hardware or operating system. The early installations were manual, requiring a large number of changes to the source and took weeks to complete (due to other higher priority problems). In fact, it was the complexity of installing the software that eventually led to the work described in this paper. More recently, (circa 1982) attempts were made to develop personal techniques and coding styles that would reduce the number of problems faced in future installations. In 1984, it became necessary to simultaneously maintain the software on two different environments (4.1bsd and System V). Furthermore, the software was not to be installed on the System V machine by its creator. Thus the evolution of a comprehensive software strategy became of utmost importance.

There are a number of ways in which this strategy might have been implemented. The particular policies that were developed as part of this work and tested by installing the 7001 software on 25 machines at 14 different sites are discussed in the following sections.

- The D-Tree
- The Magic Directory
- Source Organisation
- Tools

4.1 The D-Tree

One of the foremost problems with the early distributions was the creation of the magnetic tape containing the source, the whole source, and nothing but the source. Practically every installation prior to 1984, was delayed by the absence of necessary files (usually a file in */usr/lib* or */usr/include*) or a required modification to the UNIX system.

To resolve these problems and to achieve the isolation discussed in the previous section, it was decided that all source files (including header files) would reside in a single tree, henceforth referred to as the "D-tree". Initially, "foo" was used both as the pathname and as a proper noun to stress the importance of being able to anchor the tree at any arbitrary location. The name is still used by some, unfortunately, to differentiate between the production and research versions. All installed files (including any data or header files) would also reside in a single tree, henceforth referred to as the "dollar user IST" (or "$USRIST") tree. A single setting (in the *magic Makefile*) specifies the default location of the $USRIST tree. The default may be overridden by setting the environment variable to the desired tree (e.g., */usr/dt*).

Furthermore, changes to the "base" system (e.g., the standard UNIX tools) have been avoided when possible or, when the changes were unavoidable, installed in a non-standard location (e.g., */usr/local*). This means that the "D-tree" installation can be tested on a "vanilla" machine by excluding the updated utilities from the search path.

4.2 The Magic Directory

Another major problem faced in any installation is the establishment of the site, machine, system and configuration dependent information. In addition there are number of files (or library functions) required on certain systems but not on others (e.g., *dup2*(2)). Or a choice has to be made amongst a set of files that implement a particular facility that varies between systems (e.g., mail locking).

The main mechanism for handling this variation is to isolate the control information to a single D-tree directory, *magic*. Appendix A describes all the files of the *magic* directory and the normal installation procedure.

For a new installation, the *magic Makefile* needs to be modified as described below, and "site" and "machine" files created. The "site" file specifies information about the company and location (e.g., address, phone number) and will be normally be the same for all installations of the software within the same company. The "machine" file sets controls for the specific installation (e.g., the make pathname and the mechanisms to be used for archives, default owner and group of installed files and attributes of some of the standard installation tools).

The four lines to be changed in the "Makefile" are

```
V          = 4.3bsd
Site       = ist
Mach       = dt
USRIST     = /usr/dt
```

The "V" setting is one of the operating system names defined in the *magic sysnames.D* file which lists the names of all supported UNIX systems and associates with each a set of macros used in C and Make source to select or suppress code. The "Site" and "Mach" lines specify the site and machine information files respectively and the "USRIST" line specifies the default $USRIST setting.

In addition to the *magic* directory, the D-tree, and similar distribution trees at IST usually contain a further seven directories:

doc — reference documents and tutorials.

hdrs — the source header files, installed in $USRIST as part of the *magic* make.

install — shell scripts to do phased and environmentally pure installations.

man — the manual section tree.

src — the source for the distribution.

tools — contains a raw ("vanilla" to the extreme) make script to create the minimal set of tools needed to install the D-tree (based on the assumption that there has been no previous 7001 installation).

FL — An entire sub-system dedicated to the creation, validation and maintenance of the list of files in the D-tree.

The *FL* directory merits particular attention at this time as it won't be discussed anywhere else in the paper. The combination of the $USRIST and source tree normally consists of 33 Mega-bytes, 500 directories and 6000 files. In addition to the 3600 source and SCCS admin files, there are binaries, object files, object archives, generated header files, generated pmak scripts, diagnostic outputs, test data files and programs, ctags files, SCCS p-files, and the occasional file called *core*. To ensure the integrity of the distribution file list (not to mention meeting the restrictions imposed by the file system size) it is essential that all necessary files are properly administered and registered. The FL subsystem should be (and is) run frequently (some days almost continuously) to help ensure that the system is "clean" and complete.

4.3 Program Source Organisation

There are a number of conventions that dictate the format of standard header information in each file, as well as the manner in which files are organised in the tree.

1. All files within a single directory are "of the same type". For example, the files that make up the library routines for some program will be stored in a directory separate from the directory containing the data files. Similarly, header files will usually be in another directory.
2. All files containing code have an SCCS identification string. Thus it is possible to examine a program's binary and determine which versions of the source code were used to construct the program.
3. Each main program has a special comment line which indicates the libraries to be compiled with the program. This line, the LIBS() line, has a number of important uses (described later) and is fundamental to the operation of a number of tools. Further, it is possible to *grep*(1) the source of the program to determine the libraries it requires rather than the alternative, which is the visual examination of a (probably complicated) makefile.
4. So that programs are self-documenting, a program can always be executed to retrieve its usage. To achieve this aim, every program written at IST can be executed with a **-x** (explanation) flag. Executing a program with the "-x" flag will print its usage line and brief explanations of the program and it's flags.[7] In some cases, where the input to the program is in a fixed form, the -X flag is used to describe the input format.
5. Standard Name Conventions

   We have two purposes for using a file name convention. The first is to distinguish between true source and temporary files. The second is for the recognition of the type of a source file so that the *pmak*(1) script generators will work.

The file name conventions are not new. However, their application is consistent throughout the entire distribution. We have a standard suffix for each type of file (currently we support over 60 different file types). In addition, if for example, we have a parser that is constructed using both *Yacc*(1) and *Lex*(1), the input files are given the same prefix (e.g., **parser**). Further, if an installed C program is to be named **bill**, then the file containing the **main** function will be named **bill.c**. This file will also contain a LIBS line, which indicates any libraries to be compiled into the program.

4.4 Tools

In this section, we will describe a few of the tools (there are many others) in the D-tree. Note that a number of these programs have been developed specifically for use by other programs and some are rarely executed directly by users.

[7] We have a program, *xfinterp*(1) that, given the -x output of every program prepares it for subsequent troffing (an example is given as an appendix).

4.4.1 instal The reason behind the construction of our own version of *install*(1) is that the versions of *install*(1) vary over different systems. However, we depend upon the correct installation of programs and libraries, thus we require a version of *install*(1) that we can trust. Our program is named with only one "l" to avoid conflicts.

Our own version provides a number of important functions.

1. Flags to a particular invocation are extracted from three sources: the environment variable "$INSTFLAGS" (normally used to specify global settings such as default owner and group); the file **./Instflags.L** (used to specify special cases for specific files such as setuid modes and/or ownership selectable by system name); and on the command line (rarely used).
2. *instal*(1) provides the entry for the audit trail automatically. The importance of the audit trail has already been stressed and we believe in the importance of its construction with "no cost" to the users.
3. *instal*(1) enables a software developer to install a program whilst the program is still in use. If the binary is in use, then a copy will exist for as long as the binary is being used. All other users invoking the program will be executing the newly installed binary.

Instal(1) has 21 flags, even more than bsd's *ls*(1), but one of them is **-x** which displays a brief description of each flag. However, it is very rare to explicitly use any flags when invoking the program since they are automatically included from the INSTFLAGS environment variable or from the *Instflags.L* file. Appendix B includes the *instal*(1) **-x** output, and examples of $INSTFLAGS and a *Instflags.L* file.

4.4.2 sccs We use the *sccs*(1) interface created by Eric Allman which provides a front end to the Sccs operations. This interface has been supplemented with operations to remove and rename files. The latter two operations move the old admin files to a backup parallel directory tree, thus allowing retrieval of versions of obsolete files without cluttering the active source directories. However, the major change has been to execute the *dmail*(1) program whenever the user performs a significant operation (e.g., delta's, renames, removes or rmdel's a file).

4.4.3 dmail The *dmail*(1) tool is used to maintain the audit trail that we require. The program goes up through the directory hierarchy until a **Dmail** file is found, at which point *dmail*(1) appends a record of the change to the Dmail file. Whenever a distribution is created, a comment is added to the Dmail file. Thus all changes to the system after a distribution are easily identified.

4.4.4 mkdist This program takes as input a list of files, optionally with SCCS SID's and outputs the file contents in a number of ways. For example, it may be used to construct magnetic tape distributions or to copy files to another file system. It will search the directory tree in which the sccs rename and remove operations preserve obsolete admin files if a named file cannot be found in the active tree. Whenever a distribution is prepared a listing of all the files and their current sid's is created. This list can be used as input to mkdist to create a distribution tape or a copy of the file system.

4.4.5 filelist This is a tool that maintains a dynamic list of files. That is to say, given a file *flist* containing a list of files, *filelist*(1) may be used to set the modification time of *flist* depending upon the file list contained. A typical example is updating the *contax*(1) database, where *flist* contains a list of all the data files. *filelist*(1) is used to set *flist*'s modification time of to that of the most recently modified data file (if they all exist on the system) or to "NOW" if any of the files is missing (i.e., have been removed since the last execution of *filelist*(1)). The new, amended file list is then written back into *flist*. This program helps solve the problem of a product depending upon a changing list of files.

4.4.6 cpifdif This tool compares two files, and if they differ it copies the first file onto the second. *cpifdif*(1) is used extensively in the software construction process in order to minimise the amount of unnecessary work (it helps to circumvent one of the make time-dependency problems). Another reason for creating this as a separate tool is that *cpifdif*(1) is approximately 25 times faster than the equivalent *cmp*(1) followed by *cp*(1), does not require invoking a shell to interpret the compound command, and exits with a meaningful exit status (only non-zero if a real error occurred).

4.4.7 incls *incls*(1) is a tool that scans a file for "#include" lines and other such directives and outputs the appropriate pathnames. The manner in which the output is produced is determined by flag settings. One form of output is suitable for direct inclusion in make scripts.

4.4.8 com *com*(1) was initially developed by Tom Duff in 1976 at the University of Toronto and the "LIBS" line facility is largely an extension of his initial idea. The program extracts the first line of the argument file that contains the string "/*%" and executes the rest of that line as a shell command after replacing any embedded "%" characters by the argument file name.

We have since extended the program to process the specified (defaults to first) "/*%" or "/*@" lines. The difference between the "%" and "@" is that the embedded "@"s may be followed by special characters to indicate special strings. For example "@I" is replaced by the same string as would be generated by "InclFlags()" in pmak (i.e., the "-I..." args) and "@L" is replaced by the converted "*LIBS:" line.

5. PMAK

Whilst some problems were solved by the policies and tools thus far discussed, most had to be solved by the development of better techniques and tools for the generation and interpretation of make scripts. The major result of this effort was the *pmak*(1)[8] package as described in the following sections.

- The History and Evolution of Pmak.
- The Pmak processing.
- Pmak Preprocessor Controls and Directives.
- PMC files (pmak script generation control files).
- Pmak Script Generators.
- *pmcmd*(1) — the pmak script generator for normal program directories.
- *pmdirlist*(1) — the multi-directory pmak script generator.

5.1 The History and Evolution of Pmak

The initial research at IST on the development of better techniques to control software construction and maintenance was done by David Tilbrook and Paul Parker in 1983. In the beginning, this effort concentrated on separating the specifications of: 1) the relationship of components (analogous to a make dependency list); 2) meta-operations (make constructions not bound to specific types); and 3) type transformations resulting from bound meta-operations (the cross product of the first two parts).

[8] In footnote #1 “1,$s/\(mak\)e/p\1/g”.

This approach was taken because it was felt that some of make's problems were due to its inherent dichotomy: it tries to be both a database and a command processor, without employing a proper supporting technology. In our planned prototype, we were going to avoid this problem by using Prolog[9].

A prototype was constructed in the later stages of the project and the results did seem to suggest that a weakened form of the approach was semantically valid, however, the syntactic and preformance problems were prohibitive[10] and the work was abandoned. A full discussion of the project is not part of this paper, beyond stating that its most important contribution to our current work was to eliminate any objective other than software construction and installation.

Initially, pmak was a shell script that interpreted the argument list, used the C language preprocessor (*cpp*) to preprocess the input script (primarily to provide a proper "#include" facility), and then invoked make to interpret the output.

The *cpp* style of conditionals and macro assignments was used to deal with the differences in the versions of make and operating systems (e.g., mapping a common name to the appropriate library).

The next development was the construction of a simple script generator to handle the multi-directory make problem. This allowed us to develop a single, top level controlling make script which had a simple update and extension mechanism.

Other script generators followed to handle archive library and command directories. These replaced the previous *qed*(1) programs which were difficult to port to systems without *qed* and did not solve the problem of installing *qed* itself. These developments were facilitated by the previous development of *instal*(1) and *incls*(1).

It soon became obvious that *cpp* was inadequate, non-standard, and contained numerous bugs. In particular it treated the string "/*" as being special. Cpp is designed for use as a preprocessor for C, and as such, assumes it is safe to throw away or not process anything between the strings "/*" and the next "*/" (i.e., C comments).[11] When "/*" was required in the early version of pmak, "/?*" was used which had the same semantic interpretation as "/*" but is unacceptable. Another problem is that there is no method of commenting the input that is acceptable both to cpp and make. Furthermore, while the initial use of cpp solved some problems, it was apparent that some additional built-in facilities were desirable, to improve or facilitate the handling of the -I[12] arguments to *cc*(1) and library pathnames.

Thus *mpp*(1) was created to handle comments properly, to be software that we could distribute and to provide some additional facilities such as "#elif", expressions to test for the existence of files and built-in macros "LIBS(file)" and "InclFlags()".

Attention was then turned to the pmak shell script. At this time, the complete construction and installation of the D-tree was performed by a single pmak script that invoked pmak scripts in sub-directories. However, this used an excessive number of processes (it could wrap the process ids easily) and an excessive amount of time. Pmak was being executed 300 to 400 times per D-tree installation.

[9] There are people who claim that Prolog can do this!

[10] Is the cray free? I need to reinstall */bin/true*!

[11] The cpp "-C" flag (preserves comments for *lint*(1)) was not sufficient since the only effect was to output the comments literally.

[12] Cc "-I" flags are used to specify the directories to be searched for "#include" files. Unfortunately each directory must be specified with its own "-I" flag (i.e., separating ":"s cannot be used as in $PATH). This means that the environment variable might require embedded spaces which then raises the problem of quoting and levels of interpretation.

A version of pmak written in C was created that improved performance considerably. Also, enhancements were being made to *mpp*(1) to deal with the needs of the evolving script generators such as the addition of facilities to handle *cc*(1) "-I" flags and shell environment variables.

During the development, it became obvious that *mpp*(1) could (and had to) be incorporated into pmak, primarily so that environment variables could be set in the mpp phase and exported to the make process and to support the specification (within the input) of other application controls (e.g., the actual pathname of the application).

By this time, most of the directories contained pmak scripts that invoked a script generator to create a subscript (if it didn't already exist) and then execute pmak on the subscript. This was expensive and inconvenient.

Therefore a new mechanism, the pmak control file (known as the PMC file), was created to specify the arguments to the required script generator and pmak was modified to recreate the temporary script, *Pmak.*_[13], whenever it did not exist, was older than the PMC file or the pmak "-s" flag was specified. At this time nearly all the pmak source scripts were replaced by much shorter and simpler PMC files.

The current distribution contains five make scripts (all part of the *magic* installation and initial bootstrap which are therefore used before pmak is built), some special case Pmakfiles, and 124 PMC files averaging 4 significant lines each.

The following sections will describe some of the more important Pmak features.

5.2 The Pmak Processing

Pmak is a process that controls and prepares input for other processes. Its major role has been as a front end to make, but it can be and has been used for other applications.

Its processing can be split into three distinct phases:

I) script selection and generation (PMC processing)
II) script processing and expansion (mpp phase)
III) invocation of the application (e.g., make)

The first phase is perhaps the most interesting and novel aspect of the pmak system and enables us to achieve, with very few lines of information, a high degree of control over the construction process. Two of the five standard script generators will be discussed in later sections.

The second phase is functionally similar to cpp, but it does exhibit a number of important facilities that will be described in the next section.

5.3 Pmak Preprocessor Controls and Directives

The primary role of the mpp phase is still to provide the "#include" construct so that environmental controls and prototype make scripts could be used easily.

Most of the other facilities should be familiar to a C programmer and a listing of pmak's directives, controls, and built-in or special macros is given in Appendix C.

The following features are either novel or extremely important:

[13] The unconventional use of "_" at the end of the name *Pmak.*_ is due to the naming convention. The string "._" is used in file names to indicate that the file is temporary or generated by some process. It is normally followed by a type indicating suffix (e.g., "._h" for a generated C header file). In the case of *Pmak.*_ the prefix indicates the type thus there was no need for a suffix (and we are lazy typists).

5.3.1 InclFlags() **InclFlags()** is a built-in macro that creates "-I" flags for C compilations from the current value of the shell environment variable $INCLPATH. $INCLPATH should be a colon separated list of the directories to be searched for "#include" header files. If $INCLPATH is not set, its value is assumed to be "$USRIST/hdrs". (Recall that the default for $USRIST was set in the *magic Makefile*.)

Since one of our fundamental policies is never to change the base system, we cannot install distribution header files in a location that is normally included in the default "#include" search path. Setting

```
CFLAGS = ... InclFlags()
```

(via the inherited header file *Lcl_vars.mh*) eliminates this problem while providing a trivial relocation facility by simply changing $INCLPATH or $USRIST.

5.3.2 #inherit In addition to the "#include" facility, which is used to establish and set system wide controls, it is often desirable to override some of those settings for a particular sub-tree. For example, the destination directory for the *games* sub-tree is $USRIST/games, whereas the usual destination directory is $DESTBIN. Similarly a user altering a part of the distribution copied into their $HOME tree may wish the programs to be installed in $HOME/bin without changing any of the construction control files. It was for such cases that the "#inherit" facility was invented. The directive:

```
#inherit        Lcl_vars.mh
```

is similar to the "#include" facility in that the parameter file is incorporated into the text. However, there is an important difference. In the above example, all files called *Lcl_vars.mh* from the root ("/") down to the current directory will be included, and in that order.

For example, the files */dt/dist/Lcl_vars.mh* and */dt/dist/src/games/Lcl_vars.mh* both exist. The above "#inherit" directive will cause both files to be incorporated into pmak scripts in the games directory, with common macro settings in the latter overriding the values in the former.

In general, this facility is used to establish necessary values (e.g., $DESTBIN and $INCLPATH) at the top of the tree but to provide overrides, when necessary, at a local level.

5.3.3 LIBS() In order to implement the strategy of single sourcing and to provide a better mapping between libraries and path names, the LIBS() macro was introduced. In fact, this was one of the primary motivations for writing the preprocessor in C. The concept is simple, yet has proven to be one of the most important and powerful features of the pmak work.

The occurrence of the string "LIBS(src.c)" in pmak input is replaced by a string based on the first line, found in the file *src.c*, that contains the string "*LIBS:". When such a line is found, everything up to and including the "*LIBS:" is removed. Then, any words of the form "-lX" are converted into a string provided by pmak (i.e., */usr/lib/libcurses.a*), a string provided by the user mapping file, or to the full pathname of a file, *libX.a*, found in the normal library search path (may be extended using "$LIBPATH").

Sufficient emphasis cannot be placed upon the importance of this mechanism. The single sourcing policy is achieved since there only one legitimate place for the list of libraries, used by a program, to occur and that is at the start of the source file that contains "main()". Hence, any other programs that use the list of libraries will be using the correct values.

Using this facility, the make script dependency list for a binary contains full path names for the libraries used and the compilation statement uses the same list. The string "LIBS()" (i.e., no argument file) uses the last string generated, since the compilation statement frequently follows the dependency lines as in:

```
echo:           echo.o LIBS(echo.c)
                $(CC) $(CFLAGS) $@.o LIBS()
                mv a.out $@
```

Other programs have been modified or created to use the "LIBS" line. *com*(1) replaces "@L" in the command line by the transformation of the argument file's "LIBS" line, and a front end to *lint*(1) has been created that extracts the "LIBS" line and does a similar conversion to the appropriate lint libraries and appends the result to the end of the argument list. The latter facility means it is no longer necessary to put lint commands into make scripts thereby reducing their size and complexity.

5.3.4 Touch() In the section on make we discussed the problem of regenerating a target when the cause is not a modification to any of the files in target's dependency list. This problem cannot be solved without changing make; we have created a mechanism that provides a limited implementation of the desired facility.

The macro "Touch(prog)" is replaced by the pathname for a file *touch/prog.h* in the "#include" search path. If no file with this name is found, then the macro is replaced by the null string. For many types of target file, "Touch(prog)" (where *prog* is the tool that creates *target*) appears in the dependency list. For example:

```
target:         target.src Touch(prog)
                prog -o $@ target.src
```

If *touch/prog.h* does not exist or is older than *target* then the "Touch(prog)" has no effect. But, if we wish to force the reconstruction of all targets produced by utility *prog*, we simply *touch*(1) the *touch/prog.h* file in the search path. *touch*(1) creates its argument file if does not exist. "Touch()" is important to the management and maintenance of the distribution for two reasons:

1. the mechanism is easy to understand and use.
2. since nearly all the pmak files on our system are created by script generators, it is simple to add new "Touch()" dependencies when they are required.

Consider the awful possibility of having to force the reprocessing of all the *yacc*(1) source files on a system. On most systems this would be a major and difficult task. To date, this situation has not arisen, and we do not generate any "Touch(yacc)" dependencies. However, if it ever became necessary, the following steps would be performed:

1. alter one *printf*(3) statement in *incls*(1) to output "Touch(xxx)" where "xxx" is the appropriate processor for the suffix being processed, e.g.:

```
parsel.c:       parsel.l Touch(lex)
```

2. execute the *sh*(1) commands:

```
pmak `fnd incls`                   # instal new version of incls(1)
touch $USRIST/hdrs/touch/yacc.h    # touch the yacc Touch file
cd $FOO/src; go[14] pmak -s Instal # perform the new installation
```

The pmak "-s" flag forces all generated scripts to be recreated. Note that this

[14] *go*(1) is a program that is used to detach commands with the appropriate I/O redirections (by default into a file called *.g* after preserving old *.g* files), creates an audit trail of invocations and informs the user when the command is completed.

alteration (which will probably be in place by the time of publication) is generalised to deal with all suffices other than ".c" (felt to be just too dramatic) thus if a similar situation should arise for *lex*(1) programs, all that would be required would be the touch and pmak (without the "-s") commands.

5.3.5 IfOlder() The "IfOlder()" macro is used extensively to express dependencies in a way that can avoid the propagation of unnecessary constructions and processing. The argument is a white space separated list of files. The second and subsequent files in the list may be prefixed with by '!'. To explain the semantics consider the following example:

```
IfOlder ( f0 f1 !f2 )
```

This string will be replaced by *f0* (i.e., the first file in the list) if file *f0* does not exist, or if file *f0* is older than file *f1*, or if file *f2* does not exist. Otherwise, the macro is replaced by the null string.

As an example of its application, consider the yacc and lex source files *Fy.y* and *Fl.l*. It is assumed that *Fl.l* is the lex routine for the *Fy.y* parser and that it requires the header file *y.tab.h*[15] which contains the yacc token values.

The following pmak input is representative of every combination of lex and yacc files in the distribution and is not substantially different from the normal make constructions for such combinations.

```
Fy.c y.tab.h:       Fy.y
                    $(YACC) -d Fy.y
                    mv y.tab.c Fy.c
                    cpifdif y.tab.h Fy._h

Fy._h:              IfOlder ( y.tab.h ! Fy._h Fy.y )

Fl.o:               Fl.c Fy._h

Fl.c:               Fl.l
```

When these lines are processed by pmak, if either *y.tab.h* or *Fy._h* do not exist or if *Fy.y* is newer than *y.tab.h* the "IfOlder()" line will be transformed into

```
Fy._h:              y.tab.h
```

In any of these situations, we need the dependency illustrated above to force the recreation of the file *Fy._h*. But even then, the use of *cpifdif*(1) ensures that although *y.tab.h* is recreated, *Fy._h*'s last modification time is only changed if the contents have changed, thus we avoid an expensive recompilation of the source code output by *lex*(1). If *Fy._h* and *y.tab.h* exist and *y.tab.h* is newer than *Fy.y* then the line is replaced by

```
Fy._h:
```

and no further processing occurs.

Readers might be quick to find fault with this approach since it seems to be far too complicated to understand and use. Indeed, we would agree with them if we had been forced to write such a construction. However, the pmak script generator *pmcmd*(1) created the above construction simply because the files *Fy.y* and *Fl.l* exist, which brings us to the final aspect of pmak that will be described, script generation.

[15] We will use *y.tab.h* in the example, but in practice we rename it *Fy._H*.

5.4 PMC files

When pmak is invoked without an input script being explicitly specified, it searches the current directory for files called *Pmakfile* and *Pmak.__*. If *Pmakfile* exists, it is used as the input script. Otherwise, if *Pmak.__* exists and is newer than the *PMC* file, it is used as the input. Otherwise a new script is generated and written into the file *Pmak.__* which is then used as the input script. The pmak "-s" flag forces the *Pmak.__* file to be recreated thus may be used to override the comparison of the *PMC* and *Pmak.__* last modification times.

Script generation is controlled by the pmak control file *PMC*. This file contains a command to invoke a script generator and frequently contains the input to the generator. The command is expressed as a *com*(1) line in the *PMC* file (if the *PMC* file does not exist or does not contain a *com*(1) line then *pmcmd*(1) is used). For example, a *PMC* file in a directory of shell scripts and C programs might contain:

```
#/*% pmcmd -f %[16]
# input to pmcmd
...
```

Normally the command will be one of the standard script generators, however, any arbitrary shell command may be used. Currently all 124 PMC files in the distribution use one of the standard pmak script generators. However, before the creation of *pmproto*(1) there were quite a few PMC files that used shell scripts contained in the PMC file itself.

5.5 Pmak Script Generators

Currently there are five standard pmak script generators, whose functions are as follows:

pmcmd — to process and install xdb databases and arlo scripts, to install shell scripts, to compile and install C, Yacc, and Lex programs.

pmdirlist — to control and invoke multi-directory makes (pmaks actually).

pmlib — to build and install either a library or a program built from the library.

pmlfiles — to process and install data files and to create directories.

pmproto — to create constructions for pmak scripts according to the given prototypes. The constructs created may be dependent upon file suffices.

The following sections will describe pmcmd and pmdirlist in more detail. Examples of *pmlfiles*(1) and *pmproto*(1) PMC files are given in Appendix D.

5.6 Pmcmd

The primary purpose of *pmcmd*(1) is to process all C, Yacc, Lex, As, *Arlo*(1), *Xdb*(1), *MkHlp*(1) and Shell source files in the current directory by outputting a pmak script that will build and install the products in the appropriate location.[17] The resulting scripts will contain all the dependencies for the products by using LIBS() for libraries, *incls*(1) to generate the "#include" dependencies for C and Xdb

[16] The second "%" is replaced by the file name (i.e., *PMC*).

[17] *Arlo*(1) is a interactive application programming system. *Xdb*(1) is a system that constructs indexed databases, and is normally used to describe applications that have many component parts (e.g., xdb databases are used to describe Arlo tools, keywords, variables and error messages). *Mkhlp*(1) is a system that processes Xdb input to produce C code or a help program instead of a database.

files, and *arlouses*(1) to generate a list of Arlo sub-scripts. The Arlo and Xdb dependency lists include the appropriate "Touch()" string as described previously.

The first phase of *pmcmd*(1) processing is to create a list of source files. For a file *prog.X*, *prog* is assumed to be the name of the installed product. For example, *src.sh* is the source of the installed shell script *src*. Ambiguities are resolved by through built-in knowledge of the production steps. For example, if both *file.y* and *file.c* exist, it is assumed that *file.y* is the source and *file.c* is the *y.tab.c* file (i.e., the yacc output).

The second phase is to read the pmcmd input (if any) which may contain directives to specify a variety of controls and overrides.

Rather than explain each directive in turn we will give a contrived (though drawn, in part, from reality) and annotated example of a pmcmd input file. Comments are given after "—".

```
#/*@ pmcmd -o -f @F @r >Pmak._

— command to create the script ("-o" specifies "*.o" files are to be preserved).
— part of string after "@r" ignored by pmak and com(1) -r flag

%if unix5.?                — If the system is unix5.?
D dress[18]                — don't build or install dress
%endif

d localtool                — build localtool but don't install it.

%if *(REPL)                — If the option REPL is set
L repl rpl                 — link the installed program rpl to repl.
%endif

— "-s" flag may be specified if link to be symbolic

C parse parsey parsel      — Combine modules parse.o, parsey.o and
                           — parsel.o to produce parse

— Where yacc and lex sources are combined the output will use "IfOlder()"
— to avoid unnecessary recompilations of parsel.c.

B init /etc                — Install init in directory /etc.

— Unless overridden (as above) programs are installed in ${DESTBIN}
— and Arlo object files are installed in ${DESTLIB}/arlofiles.
```

[18] *dress*(1) is the inverse of *strip(1)*, naturally.

S bizarre — Suppress default construction for *bizarre*.

! — After "!" line the balance of the file output literally

```
bizarre:        bizarre.o LIBS(bizarre.c)
                version[19] Bizarre >relnum._h
                $(CC) $(CFLAGS) $@.o relnum.c LIBS()
                mv a.out $@
```

— the construction for *bizarre* which was suppressed by the "S" line above.

It is recognised that the directives are overly terse and could be improved substantially. There are also minor inconsistencies with the other pmak script generators that will be resolved eventually. It must be remembered that this is largely a research project and the pmak system is fundamentally a prototype. Despite these problems, most users learn to both use and understand PMC files relatively quickly, aided by the fact that all the script generators have "-X" flags that output a description of the input syntax and semantics. Most pmcmd *PMC* files (if they exist at all) are very small and the poor syntax is not a significant problem.

5.7 Pmdirlist

One of the major problems to be solved was creating a convenient and easy to use mechanism that could invoke constructions in multiple directories.

For the most part the problem seems trivial, and the need for such a tool is not obvious. However, when one is dealing with hundreds of directories, it becomes very important to be able to select subsets for construction, express the dependencies and reduce the overheads involved in changing to a new directory and invoking another pmak.

The pmak script generator *pmdirlist*(1) was developed to process a list of directories and to produce a pmak script that could be used to invoke the required operations in selected subsets of those directories. As in the previous section an annotated example of a typical "PMC" file will be used to explain and illustrate pmdirlist's input and features.

Comments are prefixed with "—".

#/*@ pmdirlist -c IL -f @F @r >Pmak._

— The omnipresent *com*(1) line also used by pmak.
— "-c IL" sets the default constructions to "Instal" and "Local"

hdrs I — *hdrs* directory to be installed but has Instal construction onl

+ libraries. — "+" lines split the input into levels

lib L hdrs — *lib* depends on *hdrs*

cmd ~ libndir lib -libist — see next paragraph

[19] *version*(1) is a program that creates an sccs id type release and version number line that may be embedded in an object file. The above construction will force the version number to be incremented on every compilation.

The "~" indicates the default constructions are used. The "-" prefix specifies dependency on either Instal or Local construction of *libist* directory (which ever is appropriate at invocation). If a prerequisite directory is suppressed (due to system selections) or the non-existence of directory, it is ignored.

Explaining pmdirlist's output is a formidable and profitless task since there are approximately 10 lines of output per directory and 40 lines per level. We will illustrate a tiny segment below, but for the most part it is not intended that pmdirlist output be read by humans or programmers.

The requirement that subsets of the constructions may be selected creates a large quantity of output. For example, nearly every directory in the distribution requires that the "hdrs" has been previously installed. But there are rarely any "hdrs" constructions to be performed and confirming that this is the case is an expensive and generally unnecessary process.

To support the above requirement, pmdirlist creates pseudo Instal and Local constructions for each level and directory, as well as additional macros, used in dependency lists. To illustrate, if pmdirlist processed the following:

```
cmd     I       dist/tools lib
```

where *dist/tools*, *lib* and *cmd* are in the first, second and third levels respectively the following would be produced:

```
#if         LVL >= 3 && Ito < 3
Local:      Local3
Instal:     Instal3
#endif

Instal3:    cmd.I
cmd.I:      L1(dist/tools) I2(lib)
            CdMake(cmd)[20] PmakFlags MakeFlags Instal

Clean::     ; CdMake(cmd) PmakFlags MakeFlags Clean
```

In pmdirlist output, "LVL" is used to specify the level up to which constructions are performed, and "Ito" is used to specify the level to which installations have been completed. The default values for "LVL" and "Ito" are 100 and 0 respectively[21].

"L1()" and "I2()" are macros whose setting depends on the value of "Ito". If "Ito" is less than **n** (where **n** is some number) "Ln(dir)" is replaced by *dir.L* (i.e., the "Local" construction for *dir*). Otherwise (i.e., Ito >= **n**) "Ln(dir)" is replaced by the null string. The same form of replacements occur for "In(dir)" except that the suffix is ".I". By specifying the values for "LVL" and "Ito" (using *cc*(1) style "-D" flags) and the arguments, it is possible to select individual directories and/or levels and to suppress levels as illustrated in the following examples:

[20] "CdMake(X)" expands to "$(MAKE) -d X" where "$(MAKE)" is set to "pmak". The "-d X" flag specifies that pmak is to "chdir" to directory "X". This facility exists so as to avoid invoking a shell process to interpret a compound command such as "cd X; exec pmak". This feature has the added benefit that a process list (that displays the argument list) will indicate which directory is being pmak'ed. The macro "PmakFlags" is the set of flags to the current pmak which will also be passed on to children pmak invocations. The macro "MakeFlags" is passed to pmak as a set of flags. However, pmak just passes these flags on to the application program (e.g., make)

[21] 32767 would have been a more logical choice for the "LVL" default, but 100 is more than sufficient. The maximum number of levels used thus far is 8 and that was to handle 50 directories. The SDI project will just have to use some other system, manually specify the setting or change pmdirlist.c.

pmak Instal	install everything
pmak -DLVL=6 Instal	install up to level 6
pmak -DIto=4 -DLVL=6 Instal	install levels 5 and 6 where up to 4 installed
pmak -DIto=4 -DLVL=6 Local3 Instal	as above and build level 3.
pmak -DIto=100 lib.L cmd.I	do Local in *lib*, Instal in *cmd*, but nothing else

It should be pointed out that most people rarely specify more than "pmak Instal". However, users who have to deal with large distributions employ these controls extensively.

6. *USE AT IST*

The techniques described above are in use at IST. They have been applied to the tools themselves and also to the ISTAR project.

Programs imported into IST are reorganised into the 7001 organisational style or modified as necessary to take advantage of the pmak package. This is usually a straightforward task which makes any subsequent development of the software easier, if only at the level of re-compiling programs after bugs have been fixed.

Perhaps the best indication of the power and usefulness of these techniques is displayed by the D-tree (wherein the tools described above are contained). There are 1400 lines of PMC, Pmakfile and Makefile[22] in 139 directories that install about 900 program and data files. Further, even though we are maintaining the source on four different machines, the PMC files are identical. Compare this with */usr/src/usr.bin* where there are some 1800 lines of make scripts with many of the prerequisites either missing or inaccurate,' in 29 directories, installing less than 80 programs on only one system with no overall controlling mechanism.

7. *CONCLUSIONS*

It would be nice to say that we are confident that the installation of the D-tree on any "sane" UNIX system will be trivial, but we cannot. Practically every installation on a new system has raised some problem. However, experience leads us to believe that the modification of the D-tree and other IST distributions to cope with the problems presented by a new system is a relatively easy task. Most portability problems can be dealt with within the existing policies and conventions. Furthermore, by virtue of the number of systems on which the software has been installed, the frequency of new problems occurring has been substantially reduced, so much so, that within one week the D-tree was installed on four different and relatively new environments with only one problem. That problem arose on the very first command of the installation and was reported with the diagnostic:

> make: not found[23]

As part of this conclusion we must list some of the short-comings and discuss the future.

The approach (and its supporting tools) is primarily the result of one person's research and is a UNIX prototype. As such, future or wider use may raise problems that have not been identified. The user community is still small and most of its applications are well controlled and usually performed by people versed in its use or with access to the creator.

[22] All of the Makefiles (300 plus lines) are used in the initial bootstrap of the software only.

[23] The site in question uses a non-native make to avoid confusion and had removed the standard "/bin/make".

The following is a list of known problems and limitations that are partially a result of these factors.

- The 7001 version of the system is constantly changing and evolving, and some of these changes require modification to the existing pmak scripts which can cause users irritation at the quarterly upgrades (though frequently not as much irritation as having the features implemented to solve their problems, though not yet available in their version).
- The approach seems overly complicated in some respects, since it reflects the complexity of the primary objective (to install over 1000 programs and data files on any UNIX system). If applied to smaller projects this complexity may raise more problems than it solves.
- The "-x" flag concept is effective only if applied to a large number of programs[24].
- The syntactic inconsistencies and obtuseness of the pmak script generators' inputs will have to be rectified if they are to be used by a larger community.
- The reorganisation of imported source done to permit the effective use of pmak, sometimes complicates the incorporation of subsequent updates from the original distributor.
- There is a loss of flexibility caused by the source reorganisation, the naming conventions and the narrow range of *pmcmd*(1) provided constructions.
- The approach can give users a false sense of security thus they fail to do adequate checking of the results of a large pmak invocation.
- There are problems ensuring that the script generators and the scripts that they produce are using the same environment, where significant.
- The addition of a new suffix to *pmcmd*(1) is non-trivial but fortunately occurs infrequently. *pmcmd*(1) should probably be converted into a *pmproto*(1) script (see appendix D). The only impediment is the differentiation between source and generated files when not distinguishable by suffix alone (e.g., *bill.y* vs. *bill.c*) and the resolution of conflicts (e.g., *fred.sh* vs. *fred.c*).
- The *instal*(1) supplementary flags file *Instflags.L* appears to have become an unnecessary luxury[25]. It would be more consistent and desirable to store special case flags in the appropriate *PMC*.

Despite these problems, the policies and tools described in this paper seem to work and we appear to have resolved most of the problems identified in section 2. Furthermore, since these policies are easier to use than to ignore, and obviously beneficial, there has been little resistance from progamming staff.

Finally, we must consider the future directions of this work. The frequency of changes to the overall strategy, policies and major tools seems to be decreasing.[26]

[24] Many of the standard tools UNIX tools, when invoked with a "-x" flag respond:

```
-x: not found
```

or perform some unknown processing, silently ignoring the specification of an unsupported flag. Unfortunately the increasing use of *getopts*(3) seems to ensure that the number of programs that mismanage arguments will rise. IST's users have had to learn that the "-x" flag may be safely applied to IST programs only.

[25] *Instflags.L* sole raison d'etre has been to reduced to permitting the direct use of *instal*(1) as a shell command (i.e., not via pmak). There are only two *Instflags.L* files left in the distribution. Both are used to specify setuid programs, and therefore the safest (and sanest) way to do the installation is via direct invocation of *instal*(1) by a super user. The D-tree has been set up to isolate all installations that require special privileges in a single *pmdirlist*(1) level.

[26] Please excuse our hysterical co-workers.

Most of the recent changes have been cosmetic or minor optimisations. It is now our opinion, that whilst the current approach is a huge improvement over those previously employed, further development would be unprofitable without advances in the following areas:

- the semantics, generation, expression, validation, control and algebras of dependencies and their extension to deal with version and environmental settings.
- an evaluation of *make*(1)'s suitability as a back-end and its simplification, enhancement and/or replacement to resolve the problems.

Currently there are a number of organisations working in the first area, using a variety of approaches. Unfortunately their objectives are widely diverging and dependent on many factors (e.g., managerial style) and there seems to be little progress or agreement. It is our belief that this is largely due to the fact that the problem is not going to be solved without experimentation and evolution, and that this cannot be done without the appropriate tools.

It is left as an exercise for the reader to guess where we will be concentrating our initial efforts.

A. The Magic Directory

The *magic* directory contains six directories and four source files. The source files are:

Makefile performs the *magic* installation.

Files.L a table of *magic* and destination pathnames. The entries of this table may be suppressed or selected depending on the system name.

Hdrs.mk a list of source header files and their "installed" pathnames (similar to *Files.L*).

sysnames.D a table of supported system names (e.g., "4.1bsd", "SUN_4.2bsd", etc.) that is used to build the header file *system.h* which contains appropriate settings and "#defines" for the selection (and suppression) of system dependent text.

The directories are:

bin contains the source and binaries of programs used in *magic* to create make scripts (e.g., to convert *Files.L* into make input) and to convert *magic* files into source and header files for the specific environment.

ext a directory of alternative implementations of facilities that vary between sites (e.g., the mail locking routines).

paths header and data files that provide pathnames or totally installation dependent settings for standard utilities (e.g., where is *chown*(1) on this system).

config header and data files that need to be set depending on the site and machine.

site_files site information files.

mach_files machine information files.

The first step in any installation is to create the site and machine files, described below.

The second step is to set four values in the *magic Makefile*. These are the name of the system (e.g., 4.3bsd), the default $USRIST value (e.g., */usr/dt*), and the suffices of the site and machine files. The installer should then check *Files.L* and the files in the *paths* directory to ensure that appropriate selections have been made. Most problems with recent installations have resulted from the failure to check these files.

Once these steps have been completed they are confirmed ("make confirm")[27] and the command "make" will use the *magic* files to install required source files in appropriate locations, to create and install the installation dependent header files, and copy the header files from their D-tree source directory to $USRIST/*hdrs*.

At this point the rest of the installation is system independent and requires no subsequent human intervention other than the use of super-user privileges to set special modes on files (a task best not done by the installation process).[28]

[27] This separate operation was instituted after an installer tried installing the D-tree on a System V machine with the *magic* settings claiming that the system was 4.3bsd.

[28] In reality the first installation on a new operating system or machine usually has one or two minor hic-cups (e.g., a "standard" header file has been renamed or is a "standard" with which we were not previously familiar). The worst case we had recently was an installation where the C compiler did not support the initialisation of auto variables. This required finding and fixing the eighty or so instances of such code. However, it is case that once the peculiarities of a machine are known subsequent upgrades or total re-installations have run flawlessly.

Any further system or site dependent code is written using the values set at this stage. For example, the following is an extract from the source for *instal*(1).

```
#include        <ist/system.h>

#if     UNIX5 /* ANON_SYS chmod(1) is buggy */
#defineBUGGY_CHMOD
#endif
...
#ifdef BUGGY_CHMOD
        /* use and check chmod(2) to circumvent chmod(1) bug */
        ...
#else
        /* use chmod(1) since it works on this system */
        ...
#endif
```

The above illustrates the "standard" manner in which such selections are made. The system setting (i.e., UNIX5) is used to suppress (or select) the definition of a another macro which, is used to suppress (or select) individual code segments. All such system dependencies have' a comment containing the string "ANON_SYS" to enable their easy location when porting to a "new" operating system.

Within shell scripts the D-tree provided program *system*(1) is used as shown:

```
case `system` in
4.?bsd|*_4.?bsd) ... ;;
*) ...
esac
```

or

```
if system "_4.?bsd" V8
then ...
else ...
fi
```

In the first use, *system*(1) outputs the name of the current system, which is used to select the appropriate "case". In the second use (i.e., with arguments), *system*(1) exits with zero status if the system name matches any of the arguments, otherwise it exits with a non-zero status.[29]

The following is the site file for the IST *magic/site_files/ist*, London office that specifies the strings used by the *magic* files to create the installation dependent header and data files.

```
# Organization profile
ORGANIZATION        Imperial Software Technology
ORGABBREV           IST
ADDRESS             60 Albert Court, Prince Consort Rd.,\
London, England\nSW7 2BH
TELEPHONE           +44 1 581 8155
TELEX               928476 istech g
```

[29] The pattern arguments to *system*(1) are the same as those used in the shell for file name expansion but with one extension: A leading "_" matches the null string or "*_" so that "_unix5.2" will match either "OSx_unix5.2" or "unix5.2" but not "punix5.2".

```
# network address and machines in same organization
SITEADDR            ist.co.uk
LCLMCHS             "ist", "istbt", "isis", "isirta"

# local time controls
GMTOFFSET           0*60
DSTTYPE             WET
TZONE               GMT
TZONEDST            BST
```

When processing an installation dependent file, names delimited by the at ('@') character are replaced by the value defined in the above file.

The following is the machine file *magic/mach_files/dt* for the research version of the D-tree, on a 4.3bsd VAX.

```
# special installation and software options list

OPTIONS             DT IST_E

GROUP               ist       # default group for installed files
OWNER               dt        # default owner for installed files

# make controls
ADDITIONAL                    # extra make settings
DORANLIB            1         # if != 0 ranlib(1) must be run on libraries
DOTSORT             0         # if != 0 tsort(1) used to order libraries
MAKEBIN             /usr/btl/bin/make5
SMARTAR             1         # if != 0 ar -xo preserves extracted file mtimes
SMARTMAKE           1         # if != 0 make supports lib.a(module.o) use

# special directory pathnames
CONTAXDIR           ~admin/contax # ~X home directory for user X
PATH                /bin:/usr/bin:/usr/ucb
PLAYDIR
SCCSBIN             /usr/btl/bin
TSDIR               %/lib/ts # % replaced by $USRIST or default
```

In the above, the "OPTIONS" setting is a list of white space separated words which is transformed into a header file that may be used in C programs to select (or suppress) features within PMC files. For example, on BLIT blessed systems, the BLIT option is added to the OPTIONS list. Then within Arlo source:

```
#if OPT_BLIT
...
#endif
```

is used to select the bit map pop-up menu code. If the directory containing the program *windsize*(1) (outputs the size of the current BLIT layer) has a PMC file containing the following lines:

```
%if ! V8(BLIT) 4.[23]bsd(BLIT)
D windsize[30]
%endif
```

then the installation of *windsize*(1) on machines other than V8 or 4.[23]bsd machines that have "BLIT" in the OPTIONS list will be suppressed. It is hoped that the other

[30] The double negative is unfortunate but necessary. Suppressing a program's installation is rare and it is assumed, that unless explicitly stated otherwise, a program is installed.

lines in the above machine file are self-explanatory, since a full explanation is beyond the scope of this paper. For a new installation, only the *magic Makefile* needs to be altered. (The site and machine files must be created for each installation and are not really part of the source except as prototypes).

The four lines to be changed are

```
V       = 4.3bsd
Site    = ist
Mach    = dt
USRIST  = /usr/dt
```

The "V" setting is one of the operating system names defined in the *magic sysnames.D* file.[31] The "Site" and "Mach" lines specify the site and machine information files respectively and the "USRIST" line specifies the default $USRIST setting.

B. Instal(1)

Flags to Instal(1).

instal [-adfilklmnqrsStxz] [-o uid] [-g gid] [-M mode] [-L link] new [old]

install file in production directory

Flags	*Explanation*
-a	list new and old file attributes
-d	mkdir necessary directories
-f	override write mode
-g gid	chgrp 'gid' new
-I	ignore chown and chgrp errors
-i	ignore flags specified in Instflags.L
-k	keep old version (implies -t)
-l	ls -l new old
-L link	ln new file to link
-m	use mv instead of cp
-M mode	chmod 'mode' new
-n	list commands but don't execute
-o uid	chown 'user' new
-q	do it quietly (see -v)
-r	ranlib new
-S	shell new in its directory
-s	strip new
-t	cp new newTMP before installing
-v	report commands (by default)
-x	display this explanation
-z	size new old

[31] When porting to a new operating system sysnames.D might have to be amended but this usually means the addition of a single line. Other source changes might be required to rectify system name based choices if the new system does not match one of the 15 or so versions of UNIX currently supported. Due to the use of ANON_SYS and *grep*(1) and *rpl*(1), finding and amending the appropriate #ifs has thus far not proven to be difficult.

Following is a typical setting for $INSTFLAGS

```
-qIta -o dt -g ist
```

Following is *Instflags.L* file for the D-tree directory containing all the special mode programs.

```
mvmail ~4.3bsd|V8
mvmail              -M 2755 -g mail
chgpw               -M 4755 -o root
```

The above specifies that on 4.3bsd and V8 the *mvmail* program requires no special settings, whereas on all other systems the flags "-M 2755 -g mail" are required. The *chgpw* installation requires "-M 4755 -o root" on all systems.

C. *List of Pmak commands and directives*

The following is a list of the *pmak*(1) commands and directives as generated by *xpmak*(1) (the *xdb*(1) database describing pmak).

#append	append a string to a macro
#define	define a macro
#elif	else if construct
#else	else part of an #if statement
#endif	end of a '#if' construct
#error	output an error message and die
#export	export a shell environment variable
#if	select or suppress lines by expression test
#ifdef	select lines if macro defined
#ifndef	select lines if macro not defined
#import	import a shell variable
#include	include named file in processing
#inherit	inherit named files
#shell	run a shell command
#undef	undefine a macro

Cwd	replaced by the current working directory
Defined	expression used to test if macro defined
Exists	expression used to test if file exists
HdrFound	expression used to test if header file found
IfOlder	replaced by first file, if older than rest of argument list or ...
InclFlags	replaced by appropriate -I flags
LIBS	replaced by libraries for argument file
Literal	replaced by literal string
MAKEBIN	name of make binary
MakeArg	expression used to test for a application argument
MakeFlags	set of special application flags passed to children
NotNil	test if string is null
PmakFlags	set of special Pmak flags passed to children
Quote	replaced by argument with suitable substitution for shell argument
Script	replaced by name of make script
ShVar	replaced by value of argument environment variable
Status	expression used to test status of last shell command
Touch	replaced by argument application touch file
UsrIst	default usrist value

D. Pmlfiles and Pmproto examples

Input to *pmlfiles*(1) (usually contained in the "PMC" file) consists of four white space separated fields ("@" to used to represent default value). The following is an annotated example.

```
#/*@ pmlfiles -f @F -d '${DEST}' @r >Pmak._
```

— Yet another com line, this one specifying the default directory is
— "'${DEST}'", quoted to delay expansion until *make* executes

```
Read_me                      — installs ${DEST}/Read_me

Makefile ${SRCDB}            — install ${SRCDB}/Makefile, 2nd field overrides default director

magicnums.D @ magicnums  — install magicnums.D as ${DEST}/magicnums

%if *(DT)
Audit @ @ -M 666
%endif
```

— on systems with DT option set install ${DEST}/Audit with mode 666

! — balance of file after "!" line output directly

A version of *pmlfiles*(1) is used in the initial magic installation phase and is still used when file naming conventions are ignored or difficult to employ. However, most uses have been replaced by application of *pmproto*(1).

pmproto processes an argument list and a prototype file, which is either the "PMC" file or one of the standard library of prototypes.

The prototype file consists of three sections (separated by "@@" lines): the prologue, argument processing, and epilogue. The prologue section is output directly. The argument section is interpreted once for each argument string. The epilogue is then output after the argument list is exhausted. An individual invocation may specify a file that is to be appended to the output script (usually when a library prototype is being used).

The following is a reduced and annotated version of the library prototype that installs TIPs *trg*(1) object files. The list of *trg*(1) source files would be specified as arguments to pmproto.

```
#include <makes/Vars.mh> — above appears in all pmak scripts to set up environment

.SUFFIXES:                 .to .t

.t.to:                     ; $(T_TRGMK) -o $@@ -p @prof@ $<
```

— addition of rules to create trg object files
— @prof@ is replaced by value as specified by "-Dprof=..." flag.

```
@@                      — start of per argument part.
— Any "@R"s replaced by the root name of the argument  (i.e., directory name and suffix
— Other such strings are supported to get the directory name ("@D"), full pathname
— ("@F"), suffix ("@S") or arbitrary basename(1) ("@B.t@") type string.

Local:                  @R.to

@R.to:                  @F @prof@ Touch(trgmk)

Instal:                 @dir@/@R.to

@dir@/@R.to:            @R.to; $(T_INSTAL) -f -M 0444 $@@
@@                      — epilogue section

Clean:                  ; -rm -f *.to ,* *._*
```

The above is a rather simple example in that it is used to process only one type of suffix (pmproto provides mechanisms to select or suppress lines on the basis of the suffix), uses the same form for all systems (the "%if" construct may be used in all the standard script generators), and does not have to process long lists of arguments.

The best indication of *pmproto*'s power and usefulness is to consider the problems of creating a make script to format, compress and install the 250 or more manual sections in "*/man/man1".

If this installation is done as a single "Instal" construct it would be far beyond the bounds of make's internal tables. The script must split the lists of files and dependencies into manageable chunks.

There are four different compression utilities: *pack*(1) on unix5 and both OSx universes; *compact*(1) on 4.1bsd; *compress*(1) on 4.[23]bsd; and *cat*(1) on all the other systems (its the only tool that can be guaranteed). Each uses a different suffix and might suppress the compression if there is no size reduction. This raises the problem of establishing which version of the installed file is required in the prerequisite list.

Despite these problems, the the man directory prototype is only 45 lines long, with 25 lines of that in the per-argument section[32]. Unfortunately, the prototype is unreadable by other than *pmproto*(1) and the author. Its use of the conditional and arithmetic expression *pmproto* facilities to split the installation into manageable parts is complicated. But using the generated scripts has prevented the execution of hundreds of unnecessary executions of *nroff*(1). Therefore the effort to create the prototype has been well rewarded.

E. Starting a C Program

Throughout this paper we have made references to a variety of components that we expect programmers to include in their source files (e.g., *com*(1) and LIBS() lines, sccsids, "-x" flag interpretation and information).

Part of our software management philosophy is that if one expects or requires certain standards to be followed then it must be simple and beneficial to do so.

[32] The generated scripts are not quite so petite. For the D-tree man1 directory the output script is approximately 3,500 lines and 70,000 characters.

In keeping with this philosophy there is a single 8 character *qed*(1) command that may be used to generate the a C program outline that contains all the required components.

In addition to the C prototype there are commands to create the shell, arlo script and xdb database outlines.

The following is an annotated and edited-to-fit example of the result of executing "\zr[33] main".

```
/*@ cc @I @F @L -o @R — the com line
— which for file fred.c and $INCLPATH xxx will become
—           cc -Ixxx fred.c /usr/lib/ist/libist.a -o fred
**
**LIBS:     -list — uses some libist.a in the LIBPATH
*/

char        Usage[] ={"? [ -x ]"}; — output in case of command syntax or semantic errors

char *      flagsexpl[] ={
"??1 line description for -x output",
"",
"-x         Display these explanations",
"-y         yet another line of -x output",
0
};

#ifndef   lint
static charsccsid[] ={"%W% - %E%"};
#endif    /* not lint */

/* created by:
**      I.M. Strange, Feb 26, 1986
**      No Such Agency (NSA)
**      Across the street from the 7/11
**      Phone:          +1 703 555 1066 (and all that)
**      Telex:          555555 nsa
**      Site:           nsa.is.uck
*/

— above generated by extracting name from the password file using usr(1),
— running fdate(1) to format the date, and
— company(1) to select and format information is supplied by the magic "site" file.
```

[33] The qed "\zr" command was initially invented by Rob Pike and David Tilbrook in 1978 when they were supposed to be building a mail system. The fundamental idea is to load a file of qed commands into a buffer, interpolate that buffer into the input stream and cleanup the mess afterwards. There are over a hundred installed "\zr" programs ranging from sccs interfaces to mechanisms for qed windows (but please remember qed is just *ed*(1) with a few minor extensions). Tom Duff was responsible for the initial development of the UNIX version of qed, but not is not to be blamed for what followed. By the way, two mail systems were eventually built: a prototype done in *qed*(1) (of course) and an mh clone in C and *sh*(1). The *qed*(1) version was more useful but not quite as easy to explain (just what does "\bm" mean??) nor to get the users to accept.

```
#include                <stdio.h>

main(argc, argv)
        char **argv;
{
        arginterp(argc, argv);
        /* you fill in this bit you poor excuse for a handball */
        — generated using insult(1)
        exit(0);
}

arginterp(argc, argv)
        char ** argv;
{
        — 30 lines of prototype argument cracker left out
}
```

F. Glossary

The following list was extracted from the "-x" flag output of D-tree programs mentioned in the paper.

apply:	apply a command to a set of arguments
arlo:	the arlo interpreter
arlouses:	list files used in an arlo script
com:	compile or process file using embedded command
company:	output company information
contax:	output selected entries from contax database
cpifdif:	compares new and old and if different copies new to old
dmail:	add argument files to Audit trail
dress:	inverse strip — puts symbol table into binary
fdate:	output formatted date string
filelist:	maintain a filelist file and its' modification times
fnd:	find a command
go:	detach a command
incls:	list include files
instal:	install file in production directory
insult:	generates an insult and disappears
ma:	The Arlo Mail System
mkhlp:	make a help facility
pmak:	preprocessed make
pmcmd:	make make script
pmdirlist:	produce master pmak script from directory dependency lines
pmlib:	output pmak script to build object archive
pmproto:	repetitively output input replacing macros by file names
qed:	somewhere between a line editor and a command interpreter
rpl:	use rep -r like output to replace lines
system:	output system name or check if argument system
tmkprof:	create new TIPs data base profile
trg:	template driven TIPs output generator
usr:	output selected /etc/passwd uid information
version:	create version string

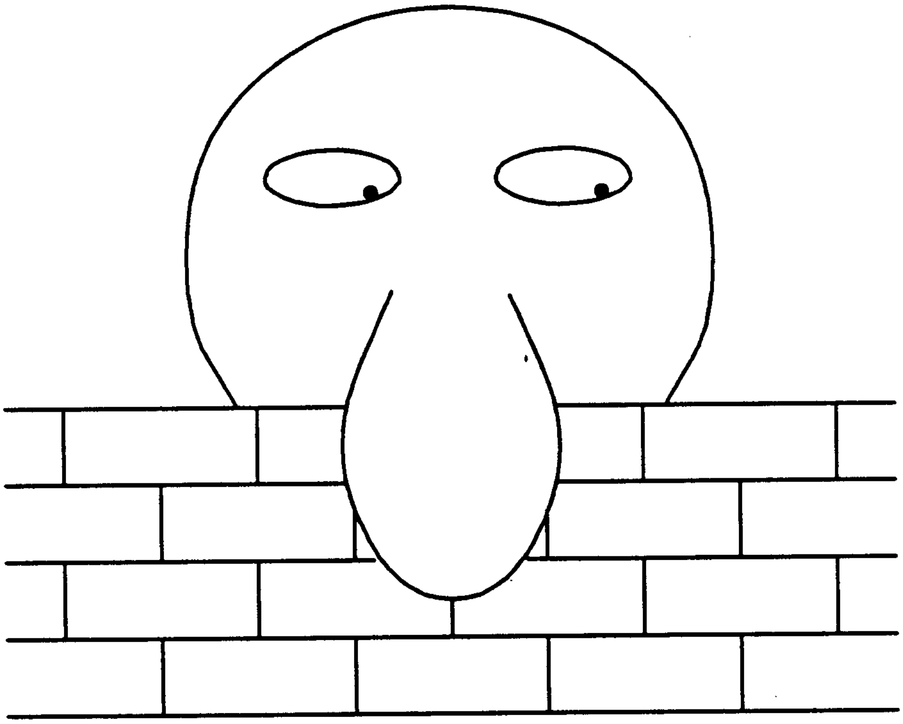
xdb: xdb database front end
xfinterp: Play with the -x flag formatting[34]
xpmak: xdb database describing the facilities and tools of the pmak system

[34] The above was generated automagically by this program and a local version of *apply*(1) using:

```
sort commands_list ¦ apply -f "%1 -x" - ¦ xfinterp -d
```

The Eighth Edition Remote Filesystem

Peter Weinberger



"Wot, No paper?"

✓

PROJECT STARGATE

Lauren Weinstein

Computer/Telecommunications Consultant
P.O. BOX 2284, Culver City, California 90231 U.S.A.

1. Project Stargate

This paper briefly outlines the aspects of an ongoing experiment into the use of satellite and cable television (CATV) technologies as a backbone for the wide dissemination of information similar to that carried by Usenet "netnews," allowing for discussions on a wide range of topics to be communicated among large numbers of individuals in distributed geographic locations. Recently, the project's scope has expanded to investigate various ways to provide greater organization for some classes of "person-to-person" electronic mail messages as well.

Due to the rapid evolution of this project and the related ongoing planning, organizational structuring, and various levels of developmental work taking place at this time, it is unfortunately not practical to do more than skim the surface of the project in this paper. In the almost two years since I first began considering the use of satellite technologies for these sorts of applications, the project has changed in both scope and form, and is now moving from being an experiment toward becoming an actual service.

In the talk that accompanies this paper, I will provide as much recent information as possible regarding project developments. Events are occurring so fast that in the two months which must elapse between the submission deadline for this paper and the actual talk, even more changes will almost certainly have occurred.

This paper assumes that the reader is already familiar with the concepts of Usenet netnews and UUCP-based electronic mail.

2. Project Stargate Background

Over the last several years, the volume of Usenet netnews traffic being carried by conventional dialup telephone and other point-to-point communication circuits has increased far beyond that originally envisioned. Not only has the number of participating Usenet sites grown enormously, but the number of articles being submitted daily into the network has also increased dramatically. Many persons feel that the overall "quality" of submitted material has also declined steadily over this period of rapid growth. Many indications point toward this traffic growth continuing at an increasing rate.

The vast majority of existing Usenet sites transfer netnews via dialup telephone connections (usually UUCP), typically at 1200 bps and sometimes at even lower speeds. The nature of the network is such that many relatively lengthy telephone calls are required, on the part of many sites, to successfully "flood" a given article throughout the network to all sites which might wish to receive it. The overall cost of these calls is not trivial and could well act as a deterrent to the successful continued operations of both Usenet and the UUCP network which underlies most of Usenet.

Even with the advent of higher speed modems, the fundamental fact remains that the transmission of netnews is inherently a "broadcast" function—that is, it is desired to get the same material to a large number of readers as quickly as possible. The point-to-point system currently being used for Usenet is simply not efficient, from either a time nor monetary standpoint, for such distributions, and threatens to become even less efficient as users and traffic continue to grow.

However, there is an alternative to multitudinous telephone calls or other point-to-point techniques for distributing netnews articles or other materials (such as mail between certain UUCP network backbone hub sites). Rather than sending such items in "conventional" manners, satellite technologies can be employed to literally broadcast information items simultaneously to most sites in the continental United States and parts of Canada via domestic communication satellites.

In the summer of 1984, with the assistance of the Usenix organization, I began a project to investigate the technical, economic, and "political" aspects of such an satellite-based distribution system. While a number of people originally predicted that nothing would come to pass from my efforts, the project and the resulting experiments have been highly successful, and are now leading toward the establishment of a generally available service in the fairly near future.

3. Vertical Interval Data

Project Stargate is based on the concept of television vertical interval data transmission. An area of the conventional television signal called the "vertical blanking interval" is mostly or completely unused by the vast majority of television broadcasters. This space can be used for the transmission of relatively high speed data given the proper equipment, without interfering with the primary video being transmitted by that signal.

Vertical interval transmission of limited capacity "magazine"-type data, with limited graphics, was pioneered by the BBC under the generic term "teletext." Teletext "magazines," providing relatively short collections of world and/or local news and other information for display directly on consumers' television screens are in use (with varying degrees of success) in a number of countries.

Unlike teletext magazines, however, Stargate is designed to send large quantities of data directly to participating sites' computers, and to allow for a high degree of user interaction with the service in much the same spirit as the existing Usenet. Stargate, by eliminating many of the delays and costs inherent in existing point-to-point distribution technologies, can be of immense benefit if it evolves properly.

A fully operational Stargate system would operate approximately as follows. Items for general distribution via Stargate (or even other messages such as certain forms of direct mail) would be sent via dialup lines as ordinary single-destination UUCP messages to a pair of dedicated microcomputers located at the satellite uplink facility of the satellite carrier. These micros would multiplex the incoming articles into a continuous data stream (complete with robust error correcting codes), possibly combine them with other data, and insert the stream into the carrier's vertical interval. The pair of microcomputers, with one acting as a "hot" standby, would help to avoid the possibility of an equipment failure causing a major disruption of the network.

4. The Stargate Experiment

The actual transmission of experimental Stargate material via satellite began a little over one year ago. I successfully installed UUCP site "stargate" on Monday, December 3rd, 1984. The Stargate computer is a Fortune 32:16 UNIX system, with a 30 megabyte disk, which Fortune Systems has graciously made available for the purposes of Stargate experimentation. Primary expenses for the experiment are currently being covered by Usenix, with special thanks due for Usenix's Lou Katz, who has strongly supported my efforts in this area through thick and thin. Thanks are also in line for Bell Communications Research (Bellcore), which has provided both moral and tangible support for the experiment. Bellcore's Brian Redman, Mike

Lesk, and Stu Feldman also deserve special thanks for their support of my efforts to make the project a reality.

Stargate is located at the primary satellite uplink facilities of Southern Satellite Systems (SSS) in a rural area about 20 miles outside of Atlanta, Georgia. The Stargate data transmissions are appearing as a portion of the vertical interval on Turner Broadcasting's "Superstation" WTBS, which is transmitted (uplinked) to satellites via SSS facilities. SSS has very generously made a continuous 1200 bps data channel available to us without charge for the experiment. This rate may increase to 2400 bps for further experiments in the near future, and may take on even higher rates at various times for an operating service. This satellite bandwidth is available to us 24 hours/day, 7 days/week and represents an enormous data transmission resource.

The WTBS signal is currently transmitted via a transponder (satellite channel) on the domestic Galaxy I satellite. WTBS is sent in this manner to almost 34 million cable TV subscriber households and businesses across the United States via nearly 10,000 cable companies. With the appropriate decoders, the Stargate transmissions are theoretically available at almost all points where satellite-delivered WTBS is received, either by cable TV or direct satellite pickup (through inexpensive earth stations) where cable delivery of WTBS is not present.

5. Receiving Stargate

We are currently in a demonstration/test mode--the exact shape of a production system is still being organized and is subject to change; a number of organizations have expressed interest in helping to create and/or support the proposed service.

Since all terrestrial points in range of the satellite transmissions receive WTBS essentially simultaneously, this is a true broadcast medium which is a considerable improvement over innumerable point-to-point phone calls or data links! The rapid dissemination of data to so many points at one time avoids one of the most serious problems with the existing Usenet, the problem of long delivery time lags, and the resulting disjoint discussions, that occur with many sites.

Full details regarding data decoder availability, costs, and various other factors are in the process of being determined. As a rule of thumb, the basic decoder "package" (commercially manufactured by a large U.S. electronics firm) will probably cost approximately $500, with another $150 to $200 or so for a special processing/buffering/interface board to handle message selection, error correction, mainframe buffering/flow control, and similar operations. The Stargate data stream continuously repeats certain portions of the overall data to ensure that all sites have a reasonable opportunity to receive the materials; the buffering board acts to handle this data stream in a manner which offloads as much processing as possible from sites' local computers. The decoder equipment includes built-in remote addressing and data decryption facilities.

It is hoped that the Stargate service will be able to allow for the rental of decoders and for the subscription of users to the information flow at a reasonable cost even if the purchase of the decoder and buffering hardware is not practical for a given end user.

A small monthly fee will presumably be assessed for Stargate data delivery in a production environment, but note that any such fee should be quite reasonable compared with the phone bills that most sites now pay, or are likely to soon pay, for large portions of conventional Usenet netnews. A fee structure for those sites unable to receive the direct satellite/cable TV transmissions but who elect to receive data from "direct" sites (via telephone connections, etc.) may also need to be established. Sites electing this non-satellite/cable TV route will generally be at a relative disadvantage since they will be unable to receive and respond to incoming materials on as timely a basis as the "direct" sites, and may find it impractical to receive all of the Stargate materials in this manner. However, for those sites which are unable to receive satellite/cable TV transmissions, other options will need to be available. The exact form of such options will depend on a variety of factors involving the structure of the Stargate organization itself and the sorts of information sent by the actual Stargate service. These issues are under study at this time.

6. Stargate Content

The issue of exactly what sort of material will be carried on a Stargate production system is a complex one, and this issue is in the process of evolving at this time. It is my personal opinion that various forms of "moderation," that is, human screening of inbound messages before they are actually sent out to the satellite, is critical to the usefulness and success of the project.

One of the serious problems with the existing Usenet is the large volume of traffic, much of which is repetitious and thus of little "useful" value. For example, even a valid technical question, asked in the proper newsgroup, may elicit many almost identical replies, all of which are costing money to transmit and, equally important, take time to read! This latter issue has become increasingly critical as the network has grown. Even if infinite amounts of data could be instantly transmitted to all points at zero cost, nobody would have the time to wade through all that material!

Already, many persons have massively cut back on the number of Usenet newsgroups they read. The main cause for this is indeed a lack of time to search through all the messages, most of which have very little real "value" (either due to repetition of previous messages' information or "poorly thought out" material) to try find the relatively few useful messages which might be buried in the mass of data!

My own view is that Stargate should function much as a highly interactive publishing/broadcasting operation. Discussion topics would be chosen from a variety of fields, and the level of moderation applied to given discussions could vary from topic to topic. For example, some discussions might need only a minimum amount of moderation to meet legal broadcast requirements and avoid repetition. Other topic discussions might be organized more along the lines of professional journals, with a much higher level of "editorial" selection exercised. The possibilities are many and varied.

7. International Issues

The Stargate satellite transmissions, being based on a U.S. domestic satellite, cannot be received outside the general area of the United States and portions of Canada. It is beyond the scope of this paper to cover the various issues relating to the possibility of international transmission of Stargate materials or the various factors which would be involved in the establishment of a similar service outside of North America. However, I'd be glad to discuss these issues with interested parties who contact me.

8. The Future

Stargate is not meant to replace the existing Usenet. The Stargate Project's plan is to provide a useful alternative to Usenet for those organizations and persons who have found the existing Usenet structure, both for technical and content reasons, to no longer fulfill all of their requirements for participatory information services. My hope is that the Stargate Project experiments will continue to evolve toward providing practical, high quality information services that can serve the needs of many persons at relatively low cost.

Getting the project to this point has certainly been fascinating. I have high hopes for the future.