EUUG

Papers presented at the
European UNIX® Systems User Group Spring Meeting

16th-18th April 1984
Nijmegen, Netherlands.

EUUG

European UNIX† Systems User Group

Proceedings EUUG Conference

Nijmegen Spring 1984

Database System Concepts &c, *Allman*	1
Using gprof to Tune the 4.2BSD Kernel, *McKusick*	10
C Unit Test Harness, *Weir*	16
Distr. Decision Making under UNIX, *Rathwell and Burns*	22
An Interactive Inf. Retrieval Syst. for UNIX, *Pell*	33
Autom. Gen. of Syntax Dir. Screen Editors, *Ridder and Florijn*	39
CRS-a Powerful Prim. for Resource Sharing in UNIX, *Nicol a.o.*	47
Honey Danber-The UUCP of the Future, *Honeyman a.o.*	56
EURIX-an UNIX based Syst. Using Eur. Nat. Languages, *Tintel*	70
Large Syst. UNIX Opportunity for Innovation, *Realini and Tucci*	74
Behind Every Binary License is the UNIX Heritage, *Redman and Parseghian*	82
Measuring Disk I/O on a VAX, *Podolski and Nei*	90
OSx: Towards a Single UNIX Syst. for Superminis, *Bott*	104
A Layered Impl. of the HP-UX Kernel..., *Lindberg*	113
An Intelligent Windowing Graphics Terminal..., *Kelley a.o.*	120
The Norwich Renal Unit Programme, *Pynsent and Pryor*	126
A Comp. of the UNIX and APSE Approaches..., *Burns and Morrison*	129

Edited by Teus Hagen, CWI, Amsterdam, Netherlands.

† UNIX is a Trademark of Bell Laboratories.

Trademarks:

UNIX is a trademark of Bell Laboratories.
VAX, PDP, Massbus, RM03 are trademarks of Digital Equipment Corporation.
IBM is a Trademark of International Business Machines.
TELETYPE is a trademark of Teletype Corporation.
Tektronix is a trademark of Tektronix, Inc.
UTS is a trademark of Amdahl Corporation.
EURIX is a trademark of Bell Telephone Mfg. Co., ITT.
WE32001 is a trademark of AT&T Technologies, Inc.
Thunderbird is a Trademark of Ford Motor Company.

Database System Concepts &c.

Eric Allman

Britton-Lee
1919 Addison, Suite 105
Berkeley, CA 94709
USA

(eric@ucbvax.UUCP, eric@Berkeley.ARPA)

ABSTRACT

Many applications maintain some long term state in the form of a database. In many cases ad hoc algorithms are sufficient (e.g., sequential scans of the password file are adequate on most systems), but often more sophisticated algorithms must be considered (e.g., mailboxes must be locked while mail is being delivered; the dictionary is too large to be practically searched sequentially).

Although ad hoc approaches are acceptable for small applications, larger applications often find it convenient to utilize a full-blown database system. Such systems may include such features as efficient access methods, logical independence from data structure, aggregation, protection, integrity constraints, multi-file capability, concurrency control, crash resilience, audit trails, and transaction control.

The structure of a database system incorporating most of these features is examined. Interfaces, data models, cost/performance tradeoffs, and the special advantages and difficulties UNIX offers to database systems are discussed.

1. DATA MODELS

It is necessary to have a model in which to consider data, that is, a structure. This structure must be general enough to handle many different data semantics, and powerful enough to be useful. There are many data models, but there are three that are both popular and sufficiently different as to be interesting: the hierarchical model, the network model, and the relational model.

1.1. The Hierarchical Model

A hierarchical model most closely approximates a tree. For example, the UNIX file system is a flexible form of hierarchy: every node in the file system has a unique parent and may have some number of children. (The UNIX file system departs from a strict hierarchical model when links are considered; files can be created that have more than one parent, i.e., may be contained in more than one directory).

Hierarchies are the most common data model. They have obvious physical correspondences, making the transition from manual to automatic systems convenient (e.g., a file cabinet is a hierarchy — a cabinet contains drawers, a drawer contains files, a file contains pages, etc.; a file can be in at most one drawer, a page in at most one file, and so forth). In the early days of computing, it was convenient that hierarchies could be represented easily on linear mediums such as punched cards or magnetic tapes.

However, hierarchies are rife with problems. Finding information when you don't know the exact file location requires a sequential scan of the entire database. Since a piece of information can

only exist in one place data must be duplicated if it could logically appear in more than one place. This technique makes updating difficult: when a piece of information is updated you must find all copies of that data.

1.2. Network Model

The network model fixes some of the problems inherent in the hierarchical model by allowing arbitrary pointers. This gives the extremely desirable property of allowing data to appear under more than one heading. For example, the description of the parts making up an assembly could reasonably appear in three places: in the file for the assembly, once in each of the files for the parts themselves, and once in each of the files of the manufacturers supplying the parts.

However, the network model also has update problems. If you include cross references (pointers), when a datum is deleted all the pointers must be found. UNIX solves this problem by deleting *references* rather than the data itself; when the last reference is deleted then the data is deleted as a side effect. In database terms this is called an "update anomaly."

Functioning network database systems (e.g., IBM's DBTG) include reference counts and back pointers so that deletion of a datum can also delete all references to that datum. The cost of this is potentially enormous. In general, network database systems are either gargantuan systems requiring huge processors or require the user to handle the "strange cases."

Hierarchies and networks share one particularly annoying property: the physical representation of the data (that is, what I can access quickly) is intertwined with the logical organization of the data. For small applications or applications that are well understood in advance (and relatively static — how many real applications do you know that fit this bill?) this is not a problem, but if the organization of the data (the *schema* in database parlance) changes, all programs accessing that data may need to be rewritten. The property we are looking for is called *data structure independence.* For example, the UNIX routines *getpwnam, et al* permit relatively trivial insertion of hashed password files, whereas the change in the directory format in 4.2bsd required changes to a number of programs that knew the physical format of the directory (but which really shouldn't have, any more than they should have had to know the number of sectors per track on the disk).

1.3. The Relational Model

A relatively recent development is the relational model. It was orignally considered little more than a mathematical curiosity, since it was "obviously" too inefficient to actually implement — much as tree-structured file systems, device independence, and dynamic processes were "obviously" too inefficient. As a result of this genesis, a large amount of the language surrounding the relational model is mathematical rather than intuitive.

In the relational model, all data is structured as a set of tables. These tables are physically unrelated to each other, although they may be logically strongly connected. For example, in the "assembly" example given above, one *relation* ("table") might contain for each assembly the list of parts that make it up; another might contain for each part the list of suppliers that supply them, etc. Connections are made using logical links: to find the list of suppliers that make parts for a given assembly, first you find the set of parts required by this assembly, and then find the list of suppliers that make those parts.

There are several important points to this example. First, the data language used to access the database is normally non-procedural (that is, it describes what data is wanted rather than how to get the data) and is set-oriented, rather than datum-oriented. Second, the relational model depends on the existence of efficient search structures. Third, key data is duplicated; the part number is listed both in the 'assembly' relation and the 'supplied-by' relation. Fourth, the data structures can be changed transparently, since the users never say "follow that pointer" in their programs.

There are several terms that merit some description:

- *relation (table, file).* A relation is a collection of semantically related information. It usually has a unique key (this is required by the mathematical model, but is frequently violated by implementations). For example, the /etc/passwd file is an example of a relation that maps login name (the unique key) to information about that user. It is sometimes called a "table" since that is a convenient printed representation for a relation.
- *tuple (record, row).* A single entry in a relation is frequently called a tuple, short for "n-tuple," from the mathematical usage. It is sometimes called a "record" (from the obvious data-processing analogy) or a "row" (from the "table" analogy).
- *attribute (field, column).* Each of the individual pieces of data in a tuple is called an attribute, once again from the mathematical model. In database-land there is nothing smaller than an attribute, since if you could subdivide an attribute you would be creating a hierarchy. In more conventional systems it would be called a "field;" the "table" analogy would call it a "column." For example, the /etc/passwd "relation" has seven attributes: user name, password, user id, group id, gcos, home directory, and shell.

2. BASIC OPERATIONS

The basic operations that are normally available in some form on all databases are:

- *retrieve (select, get).* Fetch (a) tuple(s) from the database. Conditions can normally be applied to this retrieval. For example, "retrieve all employees who earn more than their manager."
- *append (insert, add).* Add (a) tuple(s) to the database. For example, "add Eric Allman with an initial salary of $200k."
- *delete (remove).* Delete (a) tuple(s) from the database. For example, "fire everyone with salaries over $20k."
- *replace (modify, update).* Change (a) value(s) in an existing tuple(s) in the database. For example, "give all programmers a 40% raise."

In addition, other operations are implemented implicitly by some database management systems, although normally they are not given keywords in the language:

- *projection.* Select certain fields from the records, e.g., "give me names and salaries (but discard the rest of the information)."
- *restriction.* Select certain tuples from the relation, e.g., "just give me the information about the people working in software."
- *join.* Match information from one relation against another relation. For example, "match employee information against department information" (this is normally used in conjunction with restriction, so that a real query might be something like "give me employees who work in departments with sales over $1 million").
- *aggregation.* Often aggregate data is more interesting than the individual data. For example, "give me a count of employees in software" (as opposed to a list of those employees) or "what is the average salary in my company."

3. DATABASE SYSTEM STRUCTURE

Database systems are normally subdivided into a number of different functional modules. Some systems may be very strong in one area while weak in another. Since the interfaces between these modules are well-controlled, it is feasible to split a database system into different processes or processors between many of these modules.

3.1. User Interface

Most sophisticated database systems have many different user interface modules, varying from very powerful modules that require a great deal of sophistication to use effectively to modules that can be used with a minimum of training (but which have correspondingly less power).

- *Ad hoc query language* — An ad hoc query language allows a user to enter queries in a database language such as IDL or SQL. These languages require a fair amount of training to use.

  Popular languages today are non-procedural, that is, they describe the data to be accessed without describing how to find it. For example, to find the names and salaries of all employees of the toy department, one might enter (in IDL):

```
range of e is employee
range of d is department
retrieve (e.ename, e.salary)
where e.dno = d.dno
and d.dname = "toy"
```

- *Embedded programming language* — In order to build up more powerful programs, it is often popular to embed the database sublanguage into a general programming language. For example,

```
showsalaries()
{
$       char ename[50];
$       int salary;

$       range of e is employee
$       retrieve (ename = e.ename, salary = e.salary)
$       {
                printf("name=%s, salary=%d\n", name, salary);
$       }
}
```

  These interfaces require even more training.

- *Query by example* — A popular interface asks the user to fill out an example of what they would like to see. For example, if the user draws a box on the screen with columns headed "ename" and "salary" and puts "Eric Allman" in the first column and a question mark in the second column, QBE will assume that this means "give me Eric Allman's salary." Fairly minimal training is required, but complex queries are almost impossible to express.
- *Browsers* — Browsers display (usually) a single record from the database on the screen at one time. The user can, with sufficient permissions of course, update the values on the screen and then ask the database system to change the tuple accordingly. These are extremely useful for a number of common applications.

  Some browsers require that a semi-sophisticated user set up the screen format in advance, after which a naive user can use the browser. More clever browsers will set up the screen themselves, so that they can be used immediately by the naive users.
- *Application generators* — Many applications have a number of common features that are not adequately handled by a browser. An application generator provides a framework in which programs can be written. They vary from extremely simple packages to forms-based programming environments. In most cases, a medium-sophisticated user can use an applications generator. The resulting applications can generally be used by very naive users.
- *Report writers* — Businesses have a lust for reports, so naturally there is a separate class of interfaces entirely responsible for producing nicely formatted reports, with columns of figures, page headers and footers, subtotals, duplicate value supression, etc. Most report writers provide default formats that naive users can use to produce reasonably pleasing reports, with lots of hooks intelligible only to the initiated to provide fine control over the format.
- *Special purpose interfaces* — The world is filled with special-purpose interfaces. These can vary from extremely simple (e.g., Automatic Teller Machines, usable by a wholly untrained public) to extremely complex (e.g., the control program for a "factory of the future").

3.2. Decomposition and Optimization

The decomposition module is responsible for converting ("decomposing") the non-procedural query into a procedural form. The main strategies for decomposing a complex query are reduction (the "divide and conquer" technique; the query is split into two or more smaller queries, and the process is repeated recursively) and tuple substitution (bite off a small piece of the query, process that, and substitute the actual data back into the remaining query until there is nothing left; this might be called the "nibble" technique).

A good decomposition module will take the data itself into account, so as the characteristics of the data change the processing strategy will change as well. The sorts of information it will consider are:

- *Cardinality* — the number of tuples to be accessed.
- *Tuple size* — the number of bytes that the tuple occupies. Together with cardinality this can effect the number of I/O operations that must be performed.
- *Page occupancy* — how full each page of the database is on the average. It is common to not fill pages completely to allow fast insertions later.
- *Uniqueness* — are some fields of the record guaranteed to be unique? If so, certain shortcuts can be attempted. For example, when scanning the /etc/passwd file you can stop when you find the login name you are interested in, since it is "guaranteed" that the name is unique (in theory).

Aggregates (e.g., average salary) are extracted from the query, processed, and replaced back into the query, since they can be considered to be constants.

Impossible queries can be eliminated, such as asking for "salary $<$ 10000 and salary $>$ 20000." Such queries are fairly common after complex queries have been decomposed.

3.3. Execution

The real work is done in the execution module. This is often an interpreter that operates on an internal form generated by the decomposition module, although sometimes the decomposition module generates actual machine code that is loaded and executed.

Some systems such as the IDM (Intelligent Database Machine) provide special purpose hardware support that implement pieces of this module in microcode to improve performance.

3.4. Access methods

The actual structure of the data is separated from the execution module by the access methods. These know how to store data on a page so that they can be accessed efficiently. Compression (encoding of the data to minimize space consumption) is normally implemented at this level. If storage structures are available then they can minimize the searching necessary. For example, the UNIX "dbm" commands are an example of an access method.

Access methods often include caching (in a manner similar to the UNIX buffer pool) to improve performance.

Common access methods are:

- *ISAM (Indexed Sequential Access Method).* A sorted index is maintained, much like the index of a book. It can be more than one layer; for example, to find an entry in an index, you first find the correct page of the index itself (by looking at the top of the page), then find the correct column, and finally find the correct entry.

  ISAM uses a "static index" — that is, the index is not changed even if the data overflows the page. In this case, "overflow pages" are linked in, much as an update of a computer manual might have page 25, 25.1, and 25.2. If updates are common, the index can degenerate badly.
- *B-Trees.* These are trees with dynamic indicies, that is, as tuples are added the index may change structure. Although more difficult to implement than ISAM, they give better overall performance.

- *Hashing.* Hashing can be faster than either ISAM or B-trees if the exact key is known, since the index need not be searched. In practice the performance improvement is seldom critical.

  Since hashing provides an "index" that is essentially static, overflow pages must be provided. In degenerate cases performance can plummet.

  As with all hash functions, occupancy must be kept fairly low to prevent excessive collisions. As a result, storage efficiency is relatively low relative to indexed methods.

- *Unstructured.* Sometimes the cost of building an index exceeds the value of the index. This is particularly true of temporary relations, etc. Searching must be performed sequentially. The /etc/passwd file is an example of an unstructured file.

4. FEATURES AND TRADEOFFS

A number of features are available in various database systems. These features can be very important if you need them, but in virtually every case there is a cost associated with them.

4.1. Handling large databases

Grep is fine for small databases (less than a few thousand "records"). For larger databases, some sort of more powerful access methods will be required. For example, the *look* program uses a binary search algorithm on the dictionary.

4.2. Arithmetic capability inside the DBMS

It is common to need to do simple arithmetic capability inside the database management system. For example, you might need to compute "age = 1984 − birthyear" or "metres = feet * 0.3048". *Awk* is an example of a system including this capability.

4.3. Aggregation

Many applications need to know some summary of the data rather than all the values. For example, the *wc* program produces a summary of the input data. Aggregates can be simple, such as average salary or maximum age, or can return a set of values, such as total population by country (returning one value for each country in the database).

Awk includes the ability to compute these aggregates using a procedural interface.

4.4. Data structure independence

Contrary to popular belief, computer professionals are not omniscient. They often fail to properly guess (excuse me, 'understand') what the actual reference patterns will be. Data structure independence gives you the ability to change what will be most efficient to access without changing existing programs. For example, if /etc/passwd were hashed on login name and then it turned out that it would be better to produce a B-tree on user id instead, it would be nice if all the old programs still worked (probably with different performance).

This feature is common on relational systems, but rare on other types of systems.

4.5. Multi-file capability

It is often necessary to correlate data between files. This requires a more complex query language to express the queries, and of course processing is somewhat complicated. The *join* program in UNIX is an example of such a program.

4.6. Concurrent access

In commercial settings, it is common for many people to be accessing the same database at once. Some control must be provided to make sure they do not clobber one another. This is usually provided with some sort of locking mechanism. Whenever you have locking you have the potential for deadlocks, so part of the costs of this feature include the deadlock detection and resolution algorithms. For example, the UNIX mail programs must lock the mailbox during update to

insure that updates do not cause messages to overlap. Several bugs have appeared at times due to disagreements about what locking convention to use.

4.7. Crash resilience

If your data is very valuable, it is important that if the system crashes the data will be left in a consistent state. The usual definition of "consistent" is "either complete the update I was executing or back it out completely." To provide this the database system must have a notion of a "commit" operation, that is, some atomic operation that specifies that an update is to be finished rather than backed out. Since all the appropriate data must be on the disk, this implies a "sync" operation as well. Finally, all changes must be logged for the duration of the query so that they can be backed out if necessary.

4.8. Transactions (atomic multiple commands)

Often an operation that should be considered atomic must actually be implemented using several smaller operations. For example, to transfer money from one account to another each account must be updated. In the middle of the transfer, there is a brief moment when the money has either disappeared entirely or is in two accounts. Transactions can hide this window from other users, and can insure that if the system crashes at that point the database will be restored to a consistent state. This requires a more complex commit operation, and deadlocks become even more common and more difficult to resolve (since many objects may have been updated instead of just one).

4.9. Audit trails

It has been said that lawyers and bookkeepers will inherit the earth. To assuage our future owners, databases containing information of legal significance should include the ability to maintain audit trails, that is, a log of all changes (and possibly accesses) to the database. Along with satisfying our legal responsibilities, this also gives you the ability to "roll back" a database to a previous state, or, given a database dump and an audit trail (sometimes also called a transaction log), a database can be rolled forward.

4.10. Backup/recovery

Databases should be dumped periodically. A database dump is equivalent to a "level zero" dump on UNIX. In addition, a dump of the transaction log (see above) can be considered equivalent to an "incremental dump" on UNIX if the database can be rolled forward against a log.

An important issue is whether the database can be backed up while is live (active), or if it must be in a quiescent state. Some applications simply cannot afford to be offline for the several hours it can take to back up a large database.

UNIX-style dumps are not normally acceptable for databases. For example, if one record in a ten million record file changes, a UNIX incremental dump will save the entire file, while a clever database backup will save only the changes.

4.11. Protection

Often it is necessary to restrict access to data. For example, "managers can read the salaries of the people who work for them; the personnel department can read all salaries; all other access to salaries is denied." This can be relatively low resolution (e.g., access could be limited on a per-file basis as on UNIX) or very high resolution (individual fields and/or individual records protected).

4.12. Semantic integrity

Some systems give you the ability to add additional constraints on the data, for example, "salaries must be positive" or "every employee must be in a department." In fact, the use of the word "semantic" is a misnomer. The trend is toward using abstract data types.

4.13. Non-traditional data types

Typically database systems have concentrated on fairly ordinary data, such as integers, character strings, etc. As the use of databases expands they are being extended to handle new types, such as text, graphics, and "experts" (e.g., a time expert would understand "yesterday," "three weeks from last Tuesday," etc.)

5. SUITABILITY OF UNIX

5.1. Protection

The UNIX protection scheme is fairly well designed for simple protection schemes — relations (files) can be protected, as can databases (directories).

More complex protection schemes can be built easily using the set-user-id scheme.

5.2. Locking

UNIX does not contain the primitives necessary to implement locking. One technique used is to build a device driver that manages the locks; this works fine, but requires kernel modifications.

The 4.2 *flock* primitive solves this problem among cooperating processes, although the resolution (single file) is too coarse for many database applications.

5.3. File syncing

The *sync* primitive in UNIX is unacceptable to provide the assurances necessary to insure that a "commit" operation has been written to disk. First, *sync* flushes the entire cache, resulting in unacceptable performance (especially since database commits can be quite frequent). Second, when the sync primitive returns it has not guaranteed that the blocks have been written out, but only that they will be written out "soon." Obviously, I/O errors can still occur on the disk. Finally, there is no guarantee what order the blocks will be written out. If a "commit" is written out, it must first be guaranteed that all the blocks that it commits are already safely on the disk.

The 4.2bsd *fsync* primitive fixes this problem.

5.4. File system performance

The performance of the UNIX filesystem is unacceptable for large databases. This has been substantially improved in 4.2bsd. In practice this is not critical; especially large databases would normally use a raw disk anyway, since disk layout can be highly optimized.

5.5. Readahead/buffering policy

The default readahead and buffering policies that UNIX uses are almost always wrong for a database system. In particular, the database system has much more information about how the data will be used in the future. For example, during a sequential scan of a large relation old pages should be discarded immediately so that the index pages can be maintained in the cache.

This could be fixed using *ioctl* or *fcntl* "hints."

6. Summary and Conclusions

Databases are a fact of life. For example, a few of the databases used in UNIX are:

/etc/passwd
/etc/ttys
/usr/adm/wtmp
/usr/lib/aliases
/etc/fstab
/etc/termcap
/usr/dict/words
/usr/games/lib/fortune.dat

These show a large number of different access patterns. For example, /usr/dict/words is read-only, while /usr/adm/wtmp is essentially write-only.

Database systems can take care of a lot of the drudgery of computer programming, and can substantially improve the quality and performance of and application. However, a database system should be chosen carefully to find the proper balance of simplicity and growth potential. A powerful database can give you *lots* of long term flexibility, but in the short run it can cost you a great deal. If you need a database system, you should consider the following points:

- *Size* — how big is my database going to get?
- *Performance* — how fast do I have to get at the data?
- *Structure* — how complex is my data? Do I need multiple files?
- *Functionality* — what will I have to do with the data? Do I need aggregation? Computation?
- *Interfaces* — who will be accessing this data, and how do they want to see it?
- *Concurrency* — will several people be updating the data at once?
- *Dynamics* — how will my needs change in the future?

Using gprof to Tune the 4.2BSD Kernel

Marshall Kirk McKusick

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
USA

(ucbernie:mckusick)

ABSTRACT

This paper describes how the *gprof* profiler accounts for the running time of called routines in the running time of the routines that call them. It then explains how to configure a profiling kernel on the 4.2 Berkeley Software Distribution of for the VAX and discusses tradeoffs in techniques for collecting profile data. *Gprof* identifies problems that severely affects the overall performance of the kernel. Once a potential problem areas is identified benchmark programs are devised to highlight the bottleneck. These benchmarks verify that the problem exist and provide a metric against which to validate proposed solutions. Two caches are added to the kernel to alleviate the bottleneck and *gprof* is used to validates their effectiveness.

1. INTRODUCTION

The purpose of this paper is to describe the tools and techniques that are available for improving the performance of the the kernel. The primary tool used to measure the kernel is the hierarchical profiler *gprof*. The profiler enables the user to measure the cost of the abstractions that the kernel provides to the user. Once the expensive abstractions are identified, optimizations are postulated to help improve their performance. These optimizations are each individually verified to insure that they are producing a measurable improvement.

2. THE *GPROF* PROFILER

The purpose of the *gprof* profiling tool is to help the user evaluate alternative implementations of abstractions. The *gprof* design takes advantage of the fact that the kernel though large, is structured and hierarchical. We provide a profile in which the execution time for a set of routines that implement an abstraction is collected and charged to that abstraction. The profile can be used to compare and assess the costs of various implementations [Graham82] [Graham83].

2.1. Data presentation

The data is presented to the user in two different formats. The first presentation simply lists the routines without regard to the amount of time their descendants use. The second presentation incorporates the call graph of the kernel.

2.1.1. The Flat Profile

The flat profile consists of a list of all the routines that are called during execution of the kernel, with the count of the number of times they are called and the number of seconds of execution time for which they are themselves accountable. The routines are listed in decreasing order of execution time. A list of the routines that are never called during execution of the kernel is also available to verify that nothing important is omitted by this profiling run. The flat profile gives a quick overview of the routines that are used, and shows the routines that are themselves responsible for large fractions of the execution time. In practice, this profile usually shows that no single function is overwhelmingly responsible for the total time of the kernel. Notice that for this profile, the individual times sum to the total execution time.

2.1.2. The Call Graph Profile

Ideally, we would like to print the call graph of the kernel, but we are limited by the two-dimensional nature of our output devices. We cannot assume that a call graph is planar, and even if it is, that we can print a planar version of it. Instead, we choose to list each routine, together with information about the routines that are its direct parents and children. This listing presents a window into the call graph. Based on our experience, both parent information and child information is important, and should be available without searching through the output. Figure 1 shows a sample *gprof* entry.

index	%time	self	descendants	called/total called+self called/total	parents name children	index
		0.20	1.20	4/10	CALLER1	[7]
		0.30	1.80	6/10	CALLER2	[1]
[2]	41.5	0.50	3.00	10+4	EXAMPLE	[2]
		1.50	1.00	20/40	SUB1 <cycle1>	[4]
		0.00	0.50	1/5	SUB2	[9]
		0.00	0.00	0/5	SUB3	[11]

Figure 1. Profile entry for EXAMPLE.

The major entries of the call graph profile are the entries from the flat profile, augmented by the time propagated to each routine from its descendants. This profile is sorted by the sum of the time for the routine itself plus the time inherited from its descendants. The profile shows which of the higher level routines spend large portions of the total execution time in the routines that they call. For each routine, we show the amount of time passed by each child to the routine, which includes time for the child itself and for the descendants of the child (and thus the descendants of the routine). We also show the percentage these times represent of the total time accounted to the child. Similarly, the parents of each routine are listed, along with time, and percentage of total routine time, propagated to each one.

Cycles are handled as single entities. The cycle as a whole is shown as though it were a single routine, except that members of the cycle are listed in place of the children. Although the number of calls of each member from within the cycle are shown, they do not affect time propagation. When a child is a member of a cycle, the time shown is the appropriate fraction of the time for the whole cycle. Self-recursive routines have their calls broken down into calls from the outside and self-recursive calls. Only the outside calls affect the propagation of time.

The example shown in Figure 2 is the fragment of a call graph corresponding to the entry in the call graph profile listing shown in Figure 1.

The entry is for routine EXAMPLE, which has the Caller routines as its parents, and the Sub routines as its children. The reader should keep in mind that all information is given *with respect to EXAMPLE*. The index in the first column shows that EXAMPLE is the second entry in the profile listing. The EXAMPLE routine is called ten times, four times by CALLER1, and six times by

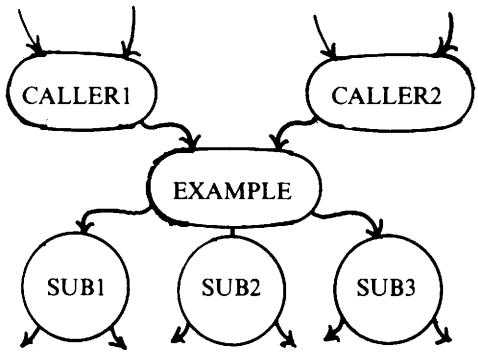


Figure 2. Example call graph fragment.

CALLER2. Consequently 40% of EXAMPLE's time is propagated to CALLER1, and 60% of EXAMPLE's time is propagated to CALLER2. The self and descendant fields of the parents show the amount of self and descendant time EXAMPLE propagates to them (but not the time used by the parents directly). Note that EXAMPLE calls itself recursively four times. The routine EXAMPLE calls routine SUB1 twenty times, SUB2 once, and never calls SUB3. Since SUB2 is called a total of five times, 20% of its self and descendant time is propagated to EXAMPLE's descendant time field. Because SUB1 is a member of *cycle 1*, the self and descendant times and call count fraction are those for the cycle as a whole. Since cycle 1 is called a total of forty times (not counting calls among members of the cycle), it propagates 50% of the cycle's self and descendant time to EXAMPLE's descendant time field. Finally each name is followed by an index that shows where on the listing to find the entry for that routine.

2.2. Profiling the Kernel

It is simple to build a 4.2BSD kernel that will automatically collect profiling information as it operates simply by specifying the **—p** option to *config*(8) when configuring a kernel. The program counter sampling can be driven by the system clock, or by an alternate real time clock. The latter is highly recommended as use of the system clock results in statistical anomalies in accounting for the time spent in the kernel clock routine.

Once a profiling system has been booted statistic gathering is handled by *kgmon*(8). *Kgmon* allows profiling to be started and stopped and the internal state of the profiling buffers to be dumped. *Kgmon* can also be used to reset the state of the internal buffers to allow multiple experiments to be run without rebooting the machine. The profiling data can then be processed with *gprof*(1) to obtain information regarding the system's operation.

A profiled system is about 5-10% larger in its text space because of the calls to count the subroutine invocations. When the system executes, the profiling data is stored in a buffer that is 1.2 times the size of the text space. All the information is summarized in memory, it is not necessary to have a trace file being continuously dumped to disk. The overhead for running a profiled system varies; under normal load we see anywhere from 5-25% of the system time spent in the profiling code. Thus the system is noticeably slower than an unprofiled system, yet is not so bad that it cannot be used in a production environment. This is important since it allows us to gather data in a real environment rather than trying to devise synthetic work loads.

3. TECHNIQUES FOR IMPROVING PERFORMANCE

This section gives several hints on general optimization techniques. It then proceeds with an example of how they can be applied to the 4.2BSD kernel to improve its performance.

3.1. Using the Profiler

The profiler is a useful tool for improving a set of routines that implement an abstraction. It can be helpful in identifying poorly coded routines, and in evaluating the new algorithms and code that replace them. Taking full advantage of the profiler requires a careful examination of the call graph profile, and a thorough knowledge of the abstractions underlying the kernel.

The easiest optimization that can be performed is a small change to a control construct or data structure. An obvious starting point is to expand a small frequently called routine inline. The drawback to inline expansion is that the data abstractions in the kernel may become less parameterized, hence less clearly defined. The profiling will also become less useful since the loss of routines will make its output more granular.

Further potential for optimization lies in routines that implement data abstractions whose total execution time is long. If the data abstraction function cannot easily be speeded up, it may be advantageous to cache its results, and eliminate the need to rerun it for identical inputs. These and other ideas for program improvement are discussed in [Bentley81].

This tool is best used in an iterative approach: profiling the kernel, eliminating one bottleneck, then finding some other part of the kernel that begins to dominate execution time.

A completely different use of the profiler is to analyze the control flow of an unfamiliar section of the kernel. By running an example that exercises the unfamiliar section of the kernel, and then using *gprof*, you can get a view of the control structure of the unfamiliar section.

3.2. An Example of Tuning

The first step is to come up with a method for generating profile data. We prefer to run a profiling system for about a one day period on one of our general timesharing machines. While this is not as reproducible as a synthetic workload, it certainly represents a realistic test. We have run one day profiles on several occasions over a three month period. Despite the long period of time that elapsed between the test runs the shape of the profiles, as measured by the number of times each system call entry point was called, were remarkably similar.

A second alternative is to write a small benchmark program to repeated exercise a suspected bottleneck. While these benchmarks are not useful as a long term profile they can give quick feedback on whether a hypothesized improvement is really having an effect. It is important to realize that the only real assurance that a change has a beneficial effect is through long term measurements of general timesharing. We have numerous examples where a benchmark program suggests vast improvements while the change in the long term system performance is negligible, and conversely examples in which the benchmark program run more slowly, but the long term system performance improves significantly.

An investigation of our long term profiling showed that the single most expensive function performed by the kernel is path name translation. We find that our general time sharing systems do about 500,000 name translations per day. The cost of doing name translation in the original 4.2BSD is 24.2 milliseconds, representing 40% of the time processing system calls, which is 19% of the total cycles in the kernel, or 11% of all cycles executed on the machine. The times are shown in Figure 3.

part	time	% of kernel
self	14.3 ms/call	11.3%
child	9.9 ms/call	7.9%
total	24.2 ms/call	19.2%

Figure 3. Call times for *namei*.

The system measurements collected showed the pathname translation routine, *namei*, was clearly worth optimizing. An inspection of *namei* shows that it consists of two nested loops. The outer loop is traversed once per pathname component. The inner loop performs a linear search through a directory looking for a particular pathname component.

Our first idea was to observe that many programs step through a directory performing an operation on each entry in turn. This caused us to modify *namei* to cache the directory offset of the last pathname component looked up by a process. The cached offset is then used as the point at which a search in the same directory begins. Changing directories invalidates the cache, as does modifying the directory. For programs that step sequentially through a directory with N files,

search time decreases from $O(N^2)$ to $O(N)$.

The cost of the cache is about 20 lines of code (about 0.2 kilobytes) and 16 bytes per process, with the cached data stored in a process's *user* vector.

As a quick benchmark to verify the effectiveness of the cache we ran "ls —l" on a directory containing 600 files. Before the per-process cache this command used 22.3 seconds of system time. After adding the cache the program used the same amount of user time, but the system time dropped to 3.3 seconds.

This change prompted our rerunning a profiled system on a machine containing the new *namei*. The results showed that the time in *namei* dropped by only 2.6 ms/call and still accounted for 36% of the system call time, 18% of the kernel, or about 10% of all the machine cycles. This amounted to a drop in system time from 57% to about 55%. The results are shown in Figure 4.

part	time	% of kernel
self	11.0 ms/call	9.2%
child	10.6 ms/call	8.9%
total	21.6 ms/call	18.1%

Figure 4. Call times for *namei* with per-process cache.

The small performance improvement was caused by a low cache hit ratio. Although the cache was 90% effective when hit, it was only usable on about 25% of the names being translated. An additional reason for the small improvement was that although the amount of time spent in *namei* itself decreased substantially, more time was spent in the routines that it called since each directory had to be accessed twice; once to search from the middle to the end, and once to search from the beginning to the middle.

Most missed names were caused by path name components other than the last. Thus Robert Elz introduced a system wide cache of most recent name translations. The cache is keyed on a name and the inode and device number of the directory that contains it. Associated with each entry is a pointer to the corresponding entry in the inode table. This has the effect of short circuiting the outer loop of *namei*. For each path name component, *namei* first looks in its cache of recent translations for the needed name. If it exists, the directory search can be completely eliminated. If the name is not recognized, then the per-process cache may still be useful in reducing the directory search time. The two cacheing schemes complement each other well.

The cost of the name cache is about 200 lines of code (about 1.2 kilobytes) and 44 bytes per cache entry. Depending on the size of the system, about 200 to 1000 entries will normally be configured, using 10-44 kilobytes of physical memory. The name cache is resident in memory at all times.

After adding the system wide name cache we reran "ls —l" on the same directory. The user time remained the same, however the system time rose slightly to 3.7 seconds. This was not surprising as *namei* now had to maintain the cache, but was never able to make any use of it.

Another profiled system was created and measurements were collected over a one day period. These measurements showed a 6 ms/call decrease in *namei*, with *namei* accounting for only 31% of the system call time, 16% of the time in the kernel, or about 7% of all the machine cycles. System time dropped from 55% to about 49%. The results are shown in Figure 5.

part	time	% of kernel
self	9.5 ms/call	9.6%
child	6.1 ms/call	6.1%
total	15.6 ms/call	15.7%

Figure 5. Call times for *namei* with both caches.

Statistics on the performance of both caches show the large performance improvement is caused by the high hit ratio. On the profiled system a 60% hit rate was observed in the system wide cache. This, coupled with the 25% hit rate in the per-process offset cache yielded an effective cache hit rate of 85%. While the system wide cache reduces both the amount of time in the routines that *namei* calls as well as *namei* itself (since fewer directories need to be accessed or searched), it is interesting to note that the actual percentage of system time spent in *namei* itself increases even though the actual time per call decreases. This is because less total time is being spent in the kernel, hence a smaller absolute time becomes a larger total percentage.

4. CONCLUSIONS

We have created a profiler that aids in the evaluation of the kernel. For each routine in the kernel, the profile shows the extent to which that routine helps support various abstractions, and how that routine uses other abstractions. The profile assesses the cost of routines at all levels of the kernel decomposition. The profiler is easily used, and can be compiled into the kernel. It adds only five to thirty percent execution overhead to the kernel being profiled, produces no additional output while the kernel is running and allows the kernel to be measured in its real environment. Kernel profiles can be used to identify bottlenecks in performance. We have shown how to improve performance by caching recently calculated name translations. The combined caches added to the name translation process reduce the average cost of translating a pathname to an inode by 35%. These changes reduce the percentage of time spent running in the system by nearly 9%.

Acknowledgements

I would like to thank Robert Elz for sharing his ideas and his code for cacheing system wide names. Thanks also to all the users at Berkeley who provided all the input to generate the kernel profiles. This work was supported by the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

References

[Bentley81] Bentley, J. L., "Writing Efficient Code", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, CMU-CS-81-116, 1981.

[Graham82] Graham, S., Kessler, P., McKusick, M., "gprof: A Call Graph Execution Profiler", Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Volume 17, Number 6, June 1982. pp 120-126

[Graham83] Graham, S., Kessler, P., McKusick, M., "An Execution Profiler for Modular Programs" Software - Practice and Experience, Volume 13, 1983. pp 671-685

[Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

C Unit Test Harness

Bill Weir

STC IDEC limited
Six Hills House
London Road, Stevenage
Herts, SG1 1YB
England

(weir@idec.UUCP)

ABSTRACT

This paper describes a Unit Test Harness developed at STC IDEC LIMITED for testing program modules written in C. A unit, or module, is defined in this context as a program component contained in a single source file; it may contain one or more functions. The Unit Test Harness (uth) is itself written in C, and runs under UNIX.

1. WHY UNIT TEST?

Modern software design methods encourage the development of highly modular programs, where each programmer is allocated one or more modules to implement. This implies a high degree of decoupling between modules; each module will, it is hoped, perform one or more identifiable operations, which may appropriately be tested in isolation from the rest of the program. It is, in any case, unlikely that monolithic testing of any non-trivial piece of software can approach the goal of 100% coverage of statements and branches: even a small module may contain many execution paths, and the total number of paths increases geometrically when modules are linked.

A 'system build' operation may be long and expensive, particularly in the case of a large embedded system, and resources will clearly be more economically used if a high level of confidence can be gained in the quality of the constituent modules before embarking on the build. Some bugs will not, of course, become apparent until the build has been carried out, but we do not want to have to rebuild and retest every time a minor module coding error is discovered.

Finally, in a cross development environment, where software development is carried out on a host mainframe or minicomputer, and executable code is downloaded on to a target microprocessor system, it may be possible to carry out functional module testing entirely on the host machine. The target system will have to be tested as a unit at some stage, but it is desirable to minimise its use in the interests of efficient resource usage.

2. UNIT TESTING DIFFICULTIES

While unit testing is a desirable goal, some difficulties must be overcome in order to achieve it.

A unit is unlikely to be self-sufficient (unless it is a single module program). It may need supporting functions, both at a higher level and at a lower level in the program structure. The first requirement we term a 'driver': i.e. a 'main' function which calls the software under test, having previously set up the environment in which it has to execute. The driver may also carry out some processing on the results returned from the software under test (even if only to display selected values on a VDU). It is unlikely that a driver written to test one module will be usable to test another - at least, not without considerable modification. The user may well be inclined to start from scratch every time a driver is required, with the resulting likelihood of introducing bugs into the drivers; this is unlikely to assist the testing process.

The latter requirement, where the software under test calls external functions which either have not yet been written, or have not yet themselves been fully tested, necessitates the creation of

'stub' functions: minimal versions of the non-existent external functions, which often do little more than just return, without error, when they are called.

Unless the software under test is a module which performs input/output functions, it will expect to acquire input data and produce output data via parameters or global variables. The driver must be able to set up the input values, and capture the outputs. Conversely, if the software under test is an i/o module, the files which it uses must be set up by the tester, or by the test driver.

These problems make it likely that, without automated module testing facilities, software developers will either fail to test at a module level at all, or will do so in an undisciplined (and undocumented) manner in which they poke data interactively at the module until they believe it to be working satisfactorily. Studies have shown that tests conducted in this fashion are seldom adequate in their coverage of program paths. It should also be noted that unless module testing is carried out against a predefined test plan, with expected results specified for every test case, its usefulness will be greatly reduced.

Regression testing, where software must be retested after modification to ensure that the modification has not produced any unfortunate side effects, is also difficult to achieve unless it is possible to repeat a set of tests consistently. Without the benefit of a testing tool which can record all tests, this necessitates (at the very least) a very large clerical task.

3. UTH FEATURES

The C Unit Test Harness addresses the above problems in a number of ways.

It runs as a batch operation, driven by a user-written test specification file. All the test data, and all information relating to the control of the tests, is localised in this one file, which can be archived as part of the program documentation. Tests may then be repeated whenever the requirement arises.

The test specification contains some syntactical constructs which are specific to uth; in general, however, these are minimised in favour of C language syntax. This means that the tester is not confronted with the necessity of learning yet another new language.

Generation of test driver modules is automatic, relieving the user of this tedious task. The current version docs not provide automatic stub generation, but it does allow the user to write his or her own stubs (in normal C source form) in a separate section of the test specification.

A test log file is produced, containing a complete record of the tests specified in the test specification. The user is required to specify predicted values for all items which are declared as output; uth checks each actual output against the corresponding predicted value, and flags an exception in the test log if they do not match. The test log can be filed along with the test specification, so that the often horrendous task of checking regression test output for consistency can be carried out quickly and easily, using standard UNIX tools.

Uth can be used in conjunction with a test coverage monitor, permitting 100% statement and branch coverage to be achieved by a fairly rapid iterative process of test specification modification and retesting.

4. UTH LIMITATIONS

In an ideal world, it would be useful to have a module testing tool which would work identically regardless of the source language in which the module is coded. This is not easy to achieve, however, and uth does not attempt it. Restriction to a single language has its own advantages, including the previously mentioned point that the language syntax can be 'borrowed'. Another point which is of some importance is that the development of uth as a usable tool was expedited considerably by the decision to restrict its use to software written in C.

The original design of uth was biased towards object oriented modules (where each module comprises a data structure, and a set of functions to operate on the data). Testing via uth is therefore almost exclusively 'black box' in nature. This means that the only items whose values may be set and examined are those which are visible (according to the C scope rules) from outside the

module under test: namely, parameters, external variables, and files. The sole exception to this rule is that 'external static' variables in the software under test (i.e. static variables defined outside the body of a function) may also be accessed.

Uth is not, in any sense, 'intelligent': it checks only the items which the user specifies as output variables, and will not therefore trap unwanted side-effects (unless every variable which may be affected is named as an output variable). It should be noted also that uth is, by its nature, restricted to the area of testing where an attempt is being made to detect flaws in program logic; it cannot handle such issues as efficiency monitoring, or concurrency.

Automatic generation of test data (to force program execution down particular paths), is a fruitful research topic, but is beyond the scope of uth at present.

Finally, uth has no interactive debug facilities. However, the executable program which it produces may be run subsequently by the user under the control of a debugger (such as 'sdb').

5. TEST SPECIFICATION FILE

The test specification file has several sections, introduced by upper case keywords. The layout is free-format.

DESCRIPTION

Free text, describing the tests which follow. C-style comments may also be used freely throughout the test specification.

ENVIRONMENT

Declarations and/or definitions of any variables which are referenced later in the test specification, as test inputs or outputs. This section may also contain the cc preprocessor directives "#include" and "#define".

FUNCTIONS

Optional section. If present, it contains the C source code of one or more functions. These may be either stub functions, or functions to perform complicated test data assignments (e.g. to initialise a large array with a set of descending integers, as test input to a sort routine).

FILEDEF

Optional section. If present, it contains brief descriptions of any files which are accessed by the module under test. In particular, it requires the user to associate a symbolic name with each such file.

CALL

Specifies the form of function call or calls by which the module under test is to be invoked for each of the following test cases.

TEST test_identifier

Associates a name ('test_identifier') with the following series of test cases: this name will appear in the test log. The user must also define in this section the variables and/or files which constitute input and output for the following test cases.

CASE case_identifier

Each CASE section (typically, several will follow each TEST section) specifies input values and predicted output values for a single invocation of the module under test. These values correspond one-for-one with the input and output item lists specified in the preceding TEST section.

The CALL, TEST and CASE sections may be repeated hierarchically: a test specification may contain one or more CALLs, each of which may be followed by one or more TESTs, each of which may be followed by one or more CASEs.

6. SPECIFICATION OF TEST DATA

Specification of test data (input and expected output) conforms broadly with the C syntax for initialisers (rather than assignments). The bracketed notation for array and structure initialisations is preserved, giving a compact format which is nevertheless familiar to the C programmer.

Uth assigns the specified inputs before each call to the module under test. On return, uth assigns the predicted outputs to its own local variables of matching type. The actual outputs are then compared against the corresponding predicted values, and any exceptions are flagged in the test log file. Floating point variables, incidentally, are a slight problem here: ideally, it should be possible for the user to specify a tolerance, within which floating point values are to be considered equal. At present, uth flags *any* discrepancy as an exception.

7. SPECIFICATION OF USER FILES

Uth regards files, in un-UNIX-like fashion, as sequences of records - where a record may be any declared data item (simple or compound). In the common case of a text file, the chosen record is most likely to be a one-dimensional character array, where the record corresponds to a line of the file. The syntax for the specification of the contents of the file is then identical to the initialisation of a two-dimensional character array :

```
{
  "first line\n",
  "second line\n",
    .
    .
    .
}
```

Uth writes the specified input data to a temporary file (unless the user has specified a path-name in the FILEDEF section). Before calling the software under test, it redirects i/o as necessary to ensure that the software under test will read the correct file, and that it will write any output to other known temporary files. On return, if any output files are specified for the current test case, uth writes the predicted output data to yet another temporary file. It then spawns a child process which runs the UNIX tool 'diff', to compare the actual and predicted files. Any output from diff is recorded in the test log file.

Binary files are processed similarly, by judicious choice of records - more than one record type may be specified for a file. The only significant difference from the treatment of text files occurs on output verification, where the actual and expected files are converted into printable form (ASCII hex), before the diff comparison.

Example

The following example is, of necessity, extremely simple, but should give the flavour of uth.

DESCRIPTION

This is a test specification for a module "multiply.c", containing a single function "multiply" (which multiplies two integer arguments and returns their product).

This test specification is in a file called "multiply.ts" (in the same directory as "multiply.c"), and the test log produced by uth execution will be in "multiply.tl".

```
ENVIRONMENT
    int   a, b, c;                  /* variable declarations */

CALL
    c = multiply(a, b);             /* function invocation   */

TEST 1
    INPUT_NAMES   a, b;         /* test case input       */
    OUTPUT_NAME   c;        /* test case output      */
    CASE 1.0      2, 3;       /* input: a=2, b=3       */
                  6;          /* expected output: c=6  */
    CASE 1.1      -4, 5;      /* input: a=-4, b=5      */
                  -20;        /* expected output: c=-20 */
```

Figure 1.

Execution of uth with the above test specification produces a test log of the following form. (Note that this example assumes a bug in the "multiply" function.)

```
UNIT TEST HARNESS (Version 1.0)

TEST LOG FILE : Thu May 3 15:30:00 1984

MODULE NAME   : multiply

TEST 1
 CASE 1.0
  Input values:
   a=2
   b=3
  Function call:
   c = multiply(a, b);
  Output values:
   c=6
 CASE 1.1
  Input values:
   a=-4
   b=5
  Function call:
   c = multiply(a, b);
  Output values:
   c=-20 (expected)  =20 (actual)  *** EXCEPTION ***
```

Figure 2.

8. UTH OPERATION

The user effort involved in using uth is solely in designing and coding the test specification: when this has been done, the tool is invoked by the command

uth modulename

Operation falls naturally into three phases:

(i) The test specification is analysed, producing three outputs:

- a test driver module (in C source form)
- a test data file (effectively identical to the tail of the test specification, from the first CALL keyword)
- a file containing the declarations of the ENVIRONMENT section

The last file is processed further by another uth component to produce a symbol table file.

(ii) In a make operation, the software under test is linked with the generated driver module, and with a library of uth utility routines, to form an executable program.

(iii) The executable program is run. It first updates the uth symbol table with the run-time addresses of the test variables, and then executes the specified test cases until the test data file is exhausted. The test log file is produced during this phase.

The components of uth are:

- a shell script of about 200 lines
- four C programs
- a library of run-time functions (also written in C)

The C components amount to 2000 - 3000 source lines in all. Initial development was done on IBM 4341 under UTS , and uth was subsequently ported to VAX , under 4.1BSD Berkeley UNIX. Development time, from conception to birth, was about nine man months (i.e. nine months of the author's time).

9. ENHANCEMENT PROGRAM

The following items, which are in no particular order, have been identified as the main desirable developments for the future.

- Remove the existing type restrictions (at present uth cannot handle typedefs or bit fields in the test specification - although there is no restriction on their use in the software under test).
- Provide a test log summariser. The test log expands rapidly if a lot of test cases are specified, obscuring the important points - i.e. the exceptions.
- Extension to handle integrations of more than one module. Although an initial design aim was that uth should be specifically a *unit* test harness, there is no reason in principle why it should not be so extended.
- Integrate uth with a test coverage monitor.
- Provide integrated debug facilities - e.g. the ability to set an execution breakpoint on a specified test case, and then use conventional symbolic debug commands to investigate the behaviour of the software under test.

References

1. D. J. Panzl: "Automatic Software Test Drivers". COMPUTER, April 1978: pp 44-50.
2. G. J. Myers: "The Art of Software Testing". J. Wiley & Sons, New York, 1979.
3. E. Miller & W. E. Howden (ed.): "Tutorial: Software Testing and Validation Techniques". IEEE, 1978.
4. M. A. Hennell: "Advanced Testing: Tools & Techniques". Liverpool Data Research Associates Ltd., 1981.

Distributed Decision Making under UNIX

M.A. Rathwell and A. Burns

Postgraduate School of Computing
University of Bradford
Richmond Rd.
W. Yorkshire
Bradford BD7 1DP
England

(maggie@ubradcs.UUCP, alan@ubradcs.UUCP)

ABSTRACT

The design and implementation of a Distributed Decision-Making (DDM) system under UNIX is described. A DDM system provides a new mechanism for linking several decision support systems (DSS) in an organisation and allows groups not necessarily linked in a hierarchical manner to cooperate with one another. The underlying structure of a DDM system consists of a community of nodes, and messages between nodes. Users can request DSS functions at other nodes and make available application tools at their nodes for calling by other users. Interdependent decision makers can semi-automate their work, explanations for decisions can be made widely available, and the resolution of conflict between nodes can be supported.

The supportive software required to implement a prototype DDM system has been built onto UNIX to construct working systems on a VAX 11/750 and a PDP 11/60. A simple general-purpose message-passing facility was developed to provide the means for inter-node communication. This has enabled experimentation with a message-based software system under UNIX and the demonstration of a variety of distributed decision-making applications.

1. INTRODUCTION

One of the growth areas in commercial computing in the last decade has been the development of decision support systems (DSS), systems which support specific decision making processes in unstructured or semi-structured problems. For example, BRANDAID is a DSS to facilitate the development of marketing plans by brand managers [1]. In a case study, Thomas and Burns [2] demonstrated the need for several linked DSS's in a manufacturing company to support organisational and group tasks which involve cooperation and conflict. Even though networking is now becoming common, a mechanism for cooperating DSS's in an organisation does not yet exist. Several people cooperating around one DSS and the distribution of a single DSS through an organisation have been described [3,4] but not interaction between several DSS's. Distributed Decision Making (DDM) is an extension of the DSS concept to allow a number of such systems to co-exist within an organisation, in a manner that need not imply any particular managerial structure.

Whereas DSS is decision-oriented and built upon management information systems (MIS), DDM is built upon the DSS layer and is concerned with organisational communication and conflict. Figure 1 shows this progression by adapting Sprague's connotational view of DSS [5]. DDM concentrates on support for and coordination amongst groups of decision makers rather than

hierarchical structuring or centralised control.

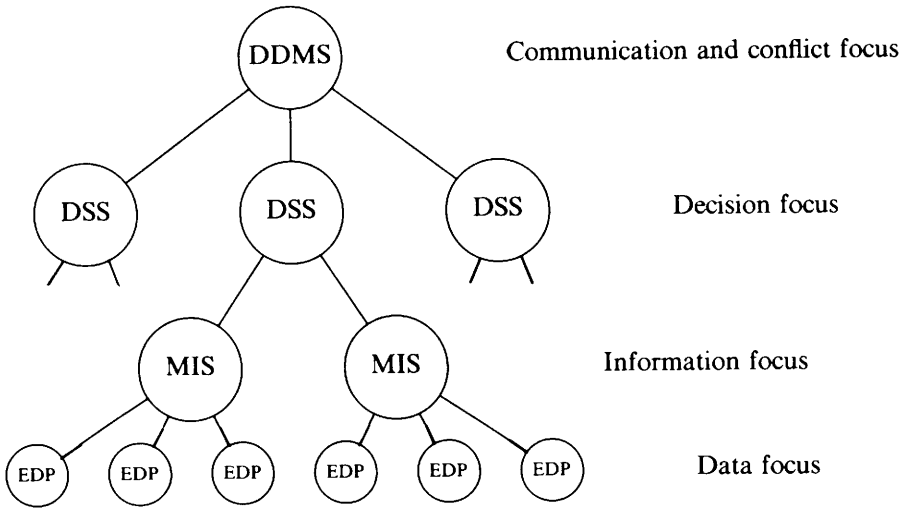


Figure 1. Focus of DDM.

Thomas and Burns [2] illustrated the need for communication and coordination in a case study of scheduling resources and manpower in two divisions of a large manufacturing company. The engineering division was responsible for the design and project management of new plant. Maintenance staff attached to the production department (electricians, plumbers etc.) were responsible for keeping existing lines working, but were also used for installing new plant as required. DSS's were built for both the engineering and maintenance groups. For example, an APL project management DSS was introduced for the project planners to support their scheduling decisions, and a DSS was developed for the tradesmen for preventative maintenance. Cooperation between the two separate DSS's, was required since both departments needed to incorporate information from each other's plans into their own DSS's.

Large organisations usually have a formal hierarchical structure with decision makers located at every level. In theory this structure enables decisions to be taken locally where possible but at a higher level when other parts of the hierarchy are involved or affected. In practice decisions have to be taken at low levels even though they may conflict with those being made at similar levels elsewhere in the organisation. This phenomenon makes it difficult to design information systems for decision makers because the problem structure cannot easily be reflected in the systems design. A DDM system is an alternative information system architecture designed to tackle this problem by concentrating on support for and coordination amongst groups of decision makers.

A DDM system allows interactions between several DSS's, each of which supports a single user or group, and provides the means to communicate relevant decisions amongst nodes. Members of a system are assumed to act as a peer group and, in the interaction of DSS's, DDM recognises differing perspectives and the inherent conflict of interests that exists within an organisation. Important features are that total decision-making participation is allowed within a structure, yet the autonomy of each unit is protected by its ability to define what the rest of the structure has access to. The support system is seen as an active element of the organisation to encourage user control of the system and an increase in individual autonomy.

2. DDM SYSTEM COMPONENTS REQUIRED

In DDM the decision-making system is viewed as a network of nodes where each node is a point in the organisation at which some decision-making activity occurs. A node may support a number of users e.g. a team or department such as in a software development project where a group want access to common data and programs. There is a need for nodes to be able to work on their own and then to join together for communication. Typically a number of DSS operations are

combined to carry out a decision making activity such as gathering information from several nodes for further analysis, for example in a planning process.

The major components of a DDM system that need to be available were outlined by Thomas and Burns [2] as follows:

(i) DSS style support for individual decision makers with operators, a database and a software interface between them and the user;

(ii) Communication between nodes, owning their own data and operators, which can negotiate for facilities;

(iii) Explanation of decisions and support for conflict resolution between nodes, probably using special purpose operators;

(iv) Evolutionary development of the whole system and facilities within it. DSS's at a node will evolve over time and nodes will be able to opt in and out of the DDM system as they wish;

(v) A flexible logical network topology which can support sequential decision making, where a decision is sent from one node to another in a required order, and group or pooled decision making with negotiation and interaction among several decision makers.

From these requirements a message-based model of the underlying structure of a DDM system, based on Hewitt's actor model of computing, was developed in order that practical systems might be constructed [6]. A description of the DDM system design and details of a prototype system implemented under UNIX follow. The prototype system is not a full implementation of the model, for example atomic actions are not supported. The aim was to obtain a simple system which could be experimented with and used to illustrate some of the main features of DDM.

3. DDM SYSTEM DESIGN

The fundamental object in a DDM system is the node, which may be a department or section in an organisation or an individual user, with some computing facility supporting a decision making activity. A DDM system consists of a loosely coupled network or community of nodes and messages between them. At each node DSS software is available to assist decision making from the perspective represented at that node. DSS facilities are generally put together as a collection of modules which we call *functions* and some of these may be made available to other nodes. A user should be able to establish a temporary connection with another node and request permission to use a function which has been declared as exportable from that node. Thus the support available includes - if permission is granted - the use of other nodes' DSS's. The programs themselves are private as is the data that these programs might use. Only the results of the execution are transmitted back to the enquirer. A *script* defines what a node should do when it receives a message. Nodes are seen as autonomous information processing units with a high degree of flexibility and control over what the rest of the system has access to.

The above logical structure implements a protocol which allows DSS's to interact. Users can combine distributed operators in a flexible way to support decision-making activities. Using *agents* - which provide a command language - models can be built which use decision support components (acting on their own databases) at other nodes. A simple example of an agent is as follows:

```
agent
  par
    sales.volume{1971, 1981}
    personnel.manpower{1971, 1981}
    sales.profit{1971, 1981}
  end par
end agent
```

Figure 2. Example agent.

The function name is prefixed by the name of the called node and a period. Any data required by the function is included in curly brackets after the function name and is passed to the node in the request message. The above agent calls the functions *volume* and *profit* at the sales node, and the function *manpower* at personnel, in each case with 1971 and 1981 supplied as data. The functions are executed in parallel. Alternatively, where data from one function call is to be used as input to another, a sequential construct is used or the function calls are nested. Agent commands are executed via an interpreter which outputs results to the screen or to a file.

The execution of a function may incorporate a dialogue with the node user. Also a function called at a node may be an agent, calling further functions or agents at other nodes. In this way a function call may initiate further actions. For example, data may be modified at each node and sent on to another node in a sequential decision-making process. The dialogue component of a DSS may also move around the system, e.g. if a node sends a questionnaire to another node invoking a function there to run the questionnaire (see Section 6). Parameters may be given to the agent interpreter to specify a predefined time period, the implication being that if it is not run in that time then the agent should be aborted as the information it is collecting is no longer useful.

For a function to be made exportable to other nodes, it must exist in some executable or interpretative form in a file held at the node, and be registered with the DDM system. Program modules can be coded in any language that the node supports. *Access rights* define to which nodes a function is exportable, and a *description* is provided to give guidance on the use of the function and any input data requirements. A function which changes the state of files at a node may be defined as *single-user* so that two or more external users are not allowed to execute the function simultaneously with the risk of multiple updates. Two functions must always be present at each node. The *help* function enables node users to find out what functions are available at a particular node and how they are called, by transmitting the names and descriptions back to the calling node. In addition *mailbox* is a permanent node function which enables users to send electronic mail to a node.

3.1. Messages

There are two basic types of messages: those requesting the execution of some program module and those answering such requests. A transaction, consisting of a request message and the associated reply, therefore resembles a remote procedure call. However agents, if they are inherently concurrent, need not wait for the reply to a message before proceeding. Messages include the name of the called node, the name of the calling node and a unique reference generated by the calling node which is used to direct the resulting data to the appropriate agent. In addition, request messages contain the name of the function to be executed and any input data necessary. Reply messages contain the status of the request, i.e. whether it was successful or not, and either data generated by the function or error messages returned from the execution. A message consists of a set of ASCII characters and can be of any reasonable length.

If a node wishes to send a message to a node whose address it does not know, then the message is sent to the *noticeboard* which is a special node incorporated within the DDM system to give increased independence to other nodes. The noticeboard must know about all nodes on the system and will send on the message to the required node. New nodes must first register with the noticeboard before they are able to receive messages. The noticeboard is the only node which should be online at all times; other nodes can, at their own discretion, withdraw from the DDM system. In doing so they can neither receive nor generate messages and so, in effect, suspend all their functions from external use. Messages for non-active nodes are initially lodged at the noticeboard and forwarded to the appropriate node, once that node becomes active again.

4. PROTOCOL SOFTWARE

In order to implement the model a core DDM system protocol was designed to allow nodes to communicate. To implement this protocol we decided to develop a message-based system under UNIX which could be experimented with. We chose the message-based system to give the flexibility and control required by the DDM system, and to show how UNIX could be used to construct a

message-based software system. However, we point out that only the model is message-based and the DDM system could be implemented in different ways.

The current prototype system, which is written in C, was first implemented on a PDP 11/60 under Version 6 UNIX, with Version 7 enhancements, and then transferred to a VAX 11/750 under Berkeley 4.1 software. Nodes are each associated with a process which runs continuously to handle the exchange of messages and control the execution of functions. These processes which are able to communicate with one another form the DDM network. The DDM system software is modular in design, with simple interaction between different parts of the system, for ease of adaptation to a physically distributed system where nodes are on different hosts.

DDM management node (DMN) software, at each node in the system, interprets DDM system commands and, where necessary, initiates transactions and releases resources. The main DMN program which runs continuously at a node is called *dmnpickup*. Only the dmnpickup process sends messages across the network. An agent requesting a function must send the request to its dmnpickup program for sending out if it is an external call. Function execution is done at the called node by dmnpickup forking a process called *dmnwait* which, if access is allowed to the calling node, forks and executes the requested function and waits for it to complete. Any input is read from the data portion of the request message. The message file descriptor (messages are implemented as files, see Section 5) is passed to dmnwait. Then, before dmnwait forks and executes the function, standard input is redirected to read from the message file, and standard output is redirected to the reply message (to capture resulting data from the function). The dmnwait program must then create a reply message containing a control flag to indicate whether or not the execution was successful, and any data generated, and send this as an internal node message to dmnpickup.

The purpose of this structure is so that dmnpickup can continue to receive and send messages instead of having to wait for function executions to complete. Thus dmnpickup forks and then continues and dmnwait is only killed off when a reply message is received after the function has executed. Dmnpickup then passes the reply message to the calling node. Replies are received by the calling node's dmnpickup and then returned to the agent.

The following diagram shows the messages involved in a DDM transaction.

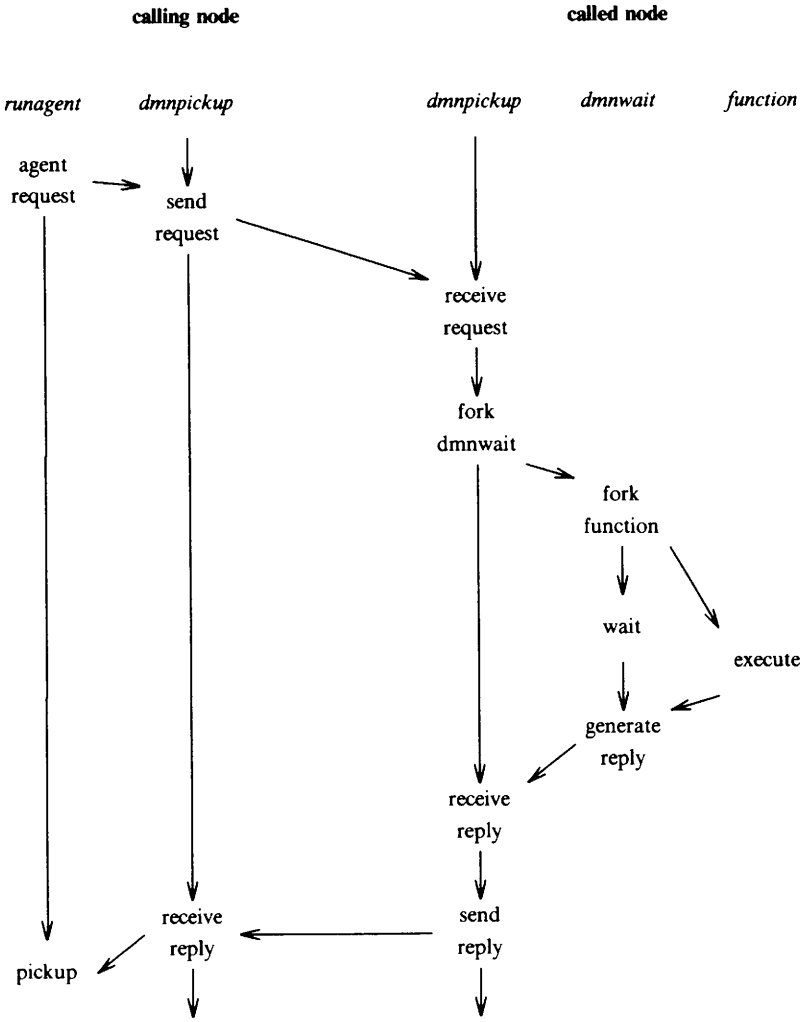


Figure 3. DDM system messages.

A distinction is made therefore between *node messages* which are internal to a node, and *system messages* which are between different nodes. Information on the handling of messages is recorded in log files at both the called and calling nodes, so that users can keep track of what functions are being requested by other nodes, and can check the progress of their own agents running in background. The log may also be used for accounting purposes. The algorithms for dmnpickup and dmnwait are as follows:

```
Do forever
    Pickup DDM message
    If system message
            If Noticeboard and redirected message
                    /* Only executed if noticeboard node */
                    If called node found
                    Resend to node
            Else if system request
                    Log enquire
                    Fork and execute dmnwait
            Else if system reply
                    Log service
                    If current agent found
                    Return reply to agent
    Else if node message
            If node request
                    Log request
                    If called node found
                    Send request to node
                    Else
                    Redirect request to noticeboard
            Else if node reply
                    Kill dmnwait
                    If calling node found
                    Send reply to node
                    Else
                    Redirect reply to noticeboard
                    Log reply.
```

Figure 4. Dmnpickup algorithm.

Note the provision for redirected messages if the node is the noticeboard.

```
Redirect standard input to read from request message
Redirect standard output to create reply message
If function exists and exportable to node
    Fork and execute function
    Wait for completion
    Receive results/status
Else
    Set status to function not exportable
Reset standard input/output
If node connected
    Send reply to node dmnpickup
Else
    Redirect reply to noticeboard
```

Figure 5. Dmnwait algorithm.

The execution of an agent is handled by an agent interpreter which generates function request messages and passes them to dmnpickup for transmission across the network. The interpreter waits for reply messages from each function requested to be received and writes resulting data to the calling node's VDU screen or a specified file. Local functions requested by an agent are passed to dmnpickup in the same way as external calls and results are passed back to the agent after execution by dmnwait. Agents can be constructed so that function calls are executed in parallel if there

is no specific order to the functions, or in sequence if, for example, data from one call is to be used as input to another. The algorithm for the agent interpreter *runagent* is as follows:

```
If node connected
    If redirect to file parameter set
            Redirect standard output
    Record agent reference in current agents
    Log agent started
    While not end of agent
            While nested function requests
                    Send last function request
                    Wait for reply
                    Return result to next function
            Send first function request
    While not end of unnested function requests
            Wait for reply
            Return results/status
    Log agent completion
    Delete agent reference from current agents
```

Figure 6. Runagent algorithm.

For nested calls runagent waits for a reply message to be received and writes the data returned from the function to the data portion of the next request message. Then for each ordinary function call and the remaining first command of embedded calls, request messages are generated until the end of the agent is reached. It is assumed that the transactions can be executed in parallel if functions are not embedded. If the node name referenced is the reserved word *all* a broadcast message is generated for each node on the system.

The node supports a command set which allows users to register functions and use the DDM system. This includes commands to add, change and delete DDM functions; to monitor agent executions and requests made on functions; and to connect and disconnect from the network. Data files required at a node includes a list of functions registered at the node containing function names, executable file names and access rights. Function descriptions are also held for use by the help facility. In addition there is a log file and a file containing the process numbers and references of currently running agents, for dmnpickup to return reply messages to. Addresses of the node's set of acquaintances are maintained but these may not be the whole system. Only the noticeboard is guaranteed to know the correct addresses of all nodes in the system. Addresses at other nodes may be incomplete or out of date.

5. MESSAGE PASSING IMPLEMENTATION

In operation of a DDM system all possible pairs of connection between nodes are needed, but not simultaneously, and connections are established temporarily. Version 6 and Berkeley 4.1 UNIX provided the facility for a number of independent processes to exist and run concurrently without mutual interference but did not provide the asynchronous communication channels required by the DDM system. A mechanism was therefore needed to implement the exchange of messages between nodes.

The main design restrictions in providing the message-passing system were that it should be easy to implement in a distributed system on several machines and that it should be as simple as possible and flexible. In the prototype DDM system which is loosely-coupled it was adequate for the message service to provide transmission of text files rather than typed data. A sophisticated interprocess communication facility such as the Carnegie-Mellon System [7] was not necessary. A simpler approach such as multiplexed files available in Version 7 UNIX was required [8].

In essence the message passing is implemented as a mailing system. A sending process transmits a message, appropriately addressed,to a receiving process; the message being added to the

pool of messages outstanding for the receiving process. If the receiving process chooses, it may take a message from its pool and process it.

The message passing was built into the operating system by adding a number of system primitives. The modifications were made first on the PDP 11/60 and then on the VAX 11/750. In all, around one hundred lines of code had to be added to the UNIX kernel to support the message passing. The message-passing primitives, which are implemented as one system call, are as follows:

```
int rxon()
int rxoff()
int nmess()
int passon(filedes, towhom)
int pickup(sender)
int *sender;
```

Figure 7. Message-passing primitives.

The sending process has only one primitive called *passon.* Passon takes two arguments: the message itself and the address of the process to receive the message. It returns an error status if there is no such address. Upon return from the primitive call the message has been despatched and the sending process continues to execute. The receiving process has two primitives available to it. The first of these is called *nmess,* and allows the process to enquire about the number of messages currently in its pool. The second primitive *pickup* allows the receiving process to take one message from the pool. Both the message itself and the address of the sender are provided to the receiving process. The identity of the sender is supplied by the message-passing system itself to allow for authentication.

If the receiving process attempts to pick a message from an empty pool, it is suspended until one arrives. There is no provision of 'interrupt' or 'signal' mechanisms to allow the receiver to be notified of the arrival of a message; it must poll the pool if it cannot be suspended using 'nmess'. In the DDM system dmnpickup loops forever waiting for DDM messages to process. For a message to be sent, the receiving process must exist, and must have previously signalled its willingness to receive messages by executing the *rxon* primitive. There is also an *rxoff* primitive which allows a process to revoke such willingness. This restriction was included to aid the debugging of software using the message passing. An attempt to pass a message to a process with permission turned off will fail. All processes start with permission set to off.

Messages consist of open UNIX file descriptors which may be passed on to cooperating processes. The 'passon' primitive takes the file descriptor and sends the indicated file to the receiving process. The sending process treats this like a close operation; for the file is disconnected from the sender. In the receiver the pickup primitive works like an open operation. A file descriptor is received and the file is then manipulated like any other.

In the current implementation, where nodes share the same host machine, no actual file transfer is undertaken. To pass on a message file all that has to be done is to rearrange pointers inside the operating system. In a physically distributed system where nodes are associated with different hosts, messages will have to be transmitted over a network or link. Messages could be broken down into packets of the maximum length permitted by the network at the sending node, and reassembled at the receiving node. The high-level DDM protocol is built on top of the message passing. The use of a high-level protocol should mean that the system can be built on top of any low-level network protocol.

Future implementations of DDM systems would use whatever interprocess communication techniques are available. System V UNIX, for example, has added message-passing system calls using a message queue set up between processes. Also a DDM system could make use of existing software support for networking to provide the connection layer underneath the DDM system, e.g. the Newcastle Connection [9]. Such software might also be used to implement the DDM protocol provided that this does not compromise the DDM system design.

6. APPLICATIONS

DDM appears to meet the need for supporting planning and decision making in a number of application areas, which involve cooperation and resource allocation tasks. Examples include engineering projects, computer hardware and software projects, research and scientific communities and company planning [10]. The DDM system implemented at Bradford is being used to support cooperation between researchers in the Computer Science Department.

A simple example of gathering information from a number of nodes in research project support is that of a reference search. Most researchers maintain a reference database at their nodes with simple software for manipulating this e.g. using the *refer* system [11]. Tools used, for example for reference searching by keywords on local files, can be registered with the DDM system for other nodes to access. Similarly simple programs for handling online papers may be used. Functions for finding text files and formatting information may be kept as local functions, but other functions, such as listing paper titles and providing copies of working papers, may be made exportable to other nodes. Other applications include project planning and sharing of software tools.

Reactive messages can be implemented using the DDM system. For example a node may send a message to another node, which automatically generates a reply message returning data to the first node. A questionnaire for collecting information can be sent to a node which invokes a function there to wake up the recipient to read it and reply. Another way we have implemented a questionnaire is by sending the instructions as a program or command file to a node for later activation. A function is called to write the instructions to a file which is then executed at the node user's convenience. The questions are responded to and a message automatically generated to send replies back to the originator. This has been used for example for scheduling meetings.

Another application is an idea dialogue or conference. Decision processes in a group often proceed in a parallel or iterative manner. For example a research problem may be broadcast to a number of individuals for comment simultaneously. A computer conferencing system has been implemented which uses the DDM system to provide asynchronous communication between conference members and keep a stored record of contributions. Also a system for generating ideas on a question or topic under debate in a group and reaching consensus has been developed using a technique called the Nominal Group Technique. This can be used for problem solving and as part of a planning process. Envisaged benefits of use of the DDM system are wider participation in planning and the generation of ideas and opinions, better coordination of work and dissemination of information, and avoidance of duplication of effort.

Currently we are investigating the programming of agents and have defined the structure of a command language for systems where distributed operators are put together to carry out tasks for the user [12]. Further work will concentrate on refining the prototype DDM system and extending it to a distributed UNIX network.

7. CONCLUSION

In conclusion the DDM system implemented meets the demands of the model by building onto the facilities provided by UNIX. There are other solutions we could have adopted. However the prototype system has proved useful for experimentation and we have been able to demonstrate a variety of distributed decision-making applications using the system. As demanded by the model, the association between nodes is not organised as a hierarchy but as communication among equals. The network structure of a DDM system provides for the communication of information and coordination and cooperation among decision makers at parallel levels. It can enhance the support that can be given to decision makers and planners whilst giving a large amount of flexibility in the way groups organise their activities.

Acknowledgements

We would like to thank Mike Banahan and Andy Rutter for their advice, and help in modifying the UNIX kernel.
The work was supported by a Science and Engineering Research Council research studentship.

References

1. Alter, S. L., *Decision support systems : current practices and continuing challenges,* Addison Wesley (1980).
2. Thomas, R. C. and Burns, A., "The Case for Distributed Decision Making Systems," *Computer Journal* **25**(1) pp. 147-152 (1982).
3. Keen, P. G. W. and Hackathorn, R. D., "Decision Support Systems and Personal Computing," Sloan School of Management Working Paper 1088-79, Cambridge, Massachusetts (1979).
4. Scher, J. M., "Distributed Decision Support Systems for Management and Organisations," *First International Conference on Decision Support Systems*, pp. 130-140 Execucom Systems Corporation, (June 8-10 1981).
5. Sprague, R. H. and Carlson, E. D., *Building effective decision support systems,* Prentice-Hall, New Jersey (1982).
6. Rathwell, M. A., Burns, A., and Thomas, R. C., "Modelling Distributed Decision-Making Systems," *Computer Centre Research Report, CCR.10*, University of Bradford, (1983).
7. Rashid, R., "An Interprocess Communication Facility for UNIX Version 7," CMU-CS-80-124, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pennsylvania (February 1980).
8. Rosenthal, D. S. H., "A Survey of Asynchronous I/O Techniques for UNIX," Report 81/13, EdCAAD Studies, Dept. of Architecture, Edinburgh University (November 30, 1981).
9. Brownbridge, D. R., Marshall, L. F., and Randell, B., "The Newcastle Connection or UNIXes of the World Unite!," *Software - Practice and Experience* **12** pp. 1147-1162 (December 1982).
10. Rathwell, M. A. and Burns, A., "Information System Support for Group Planning and Decision-Making Activities: An Analysis of the Applicability of the Distributed Decision-Making Approach," *Computer Centre Research Report, CCR.13*, University of Bradford, (1983).
11. Lesk, M. E., "Some Applications of Inverted Indexes on the UNIX System," Bell Laboratories Computing Science Technical Report 69 (June 1978).
12. Rathwell, M. A. and Burns, A., "AL: A Command Language for Distributed Support Systems," *Computer Centre Research Report, CCR.48*, University of Bradford, (1984).

An Interactive Information Retrieval System for UNIX

A. R. Pell

Dept. of Computer Science
University of Reading
Whiteknights Park
Reading RG6 2AX
England

(adrian@ru-cs44.UUCP)

ABSTRACT

Many problems exist for which an appropriate solution is an information retrieval system. They may be used, for instance, to maintain address lists, skill registers, library catalogues, etc.

This paper describes an implementation of such a system. Emphasis has been laid throughout on providing a flexible system suited to the needs of a novice user.

Observations are also presented on different forms of user interface to such a system.

1. INTRODUCTION

Office and administrative systems are frequently used to manipulate large quantities of data. This task is often considered to be a secretarial task, and thus traditional UNIX tools are not necessarily suitable as the user may be inexperienced in the operation of the powerful facilities available.

Systems designed for a novice user need to present a clear, consistent view of the task in hand and to present no surprises. The term *user-friendly* is frequently used to describe systems with these characteristics.

This paper describes the design and implementation of a user-friendly interactive information retrieval system. Where possible, existing UNIX facilities have been used to speed implementation. Whilst this has not been ideal, the use of these facilities is not so intimately bound to the system that they may not be exchanged, in the future, for some more suitable realisation. In particular, the input and output modules will probably be changed at a later stage of the project to allow use of differing input devices.

The user interface to this system has, therefore, been kept intentionally simple and uncluttered. Whilst further facilities may be added at a later stage, it must still be possible for the user to maintain a picture of the overall system.

2. DESIGN

The project was sub-divided into a number of parts:

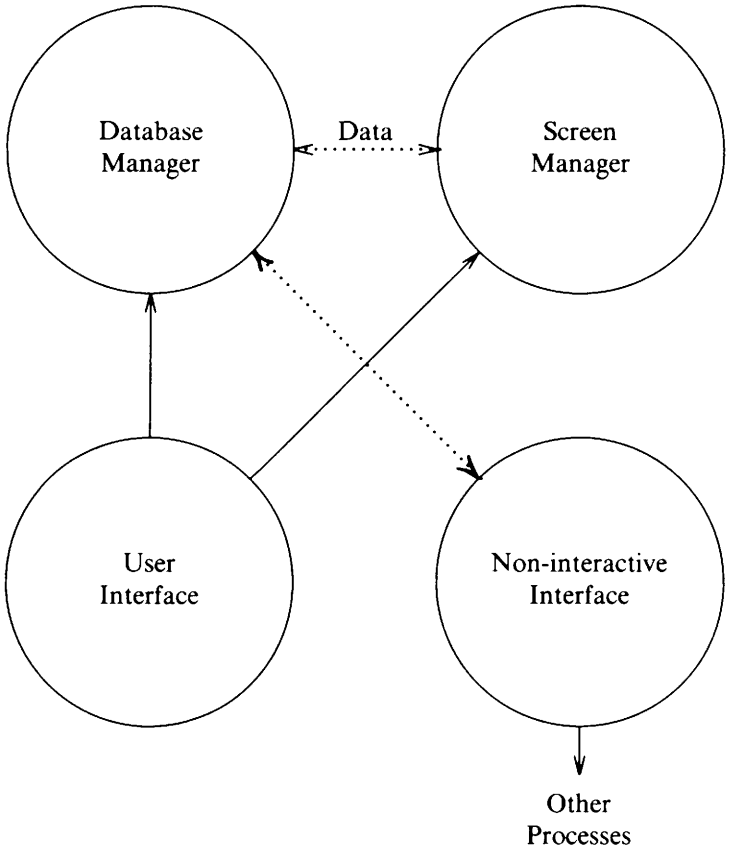
i) *database management* - responsible for the safe storage and retrieval of data items,

ii) *on-screen data manipulation* - displays items to be modified in a form suitable for alteration by the user

iii) *user interaction* - handles all status reports and user prompting in a uniform manner

iv) *non-interactive interface* - a mechanism for extracting information from the database in a consistent manner for presentation to other processes.

These sections are linked together in the following fashion:



In this section we look at the design issues affecting each part.

2.1. Database management

The database management module is required to provide access to a number of separate information bases. The number of such information bases is not necessarily known in advance, nor are the particular characteristics of items contained in them. Thus, a flexible interface is required allowing specification and modification of the data being handled.

Each database consists of a number of records each representing a single item of information of interest to the user. These records consist of a number of fields, the maximum number being determined by the implementation of the database management module.

Each field of a record may take one of the following forms:

- a character string, the length of which may be specified in advance
- a number, real or integer
- a date
- a time
- a combined date and time.

In addition to specified fields, each record has added to it (transparently to the user), the time of last update and the user id of the updater. This information may be used at some stage in the future to provide a limited amount of error recovery or accounting. At present, the last information available is displayed along with associated record. A unique serial number is also added to each record. This may be used to identify records which are candidates for deletion or updating.

The essential capabilities required of the database management module are:

- add a new record
- selectively retrieve records
- selectively delete records
- selectively modify records

2.2. On-screen data manipulation

Items are displayed on screen for one of two basic reasons:

i) they have been selected for viewing, modification or deletion by the user on the basis of specification of values for a number of fields.

ii) a new item is being entered by the user.

In both cases, the problems are the same: the current state of the item must, at all times, be maintained in a consistent fashion, th user must be able to move freely around the item to examine and, if appropriate, change various of the fields.

A *form* is a structure containing all necessary information about an item in the database to allow it to be displayed and, if necessary, changed. This must include not only the current values of each field in the record but its current screen location (if any) to allow the form to be manipulated on screen.

All data within the system are manipulated as forms.

Initially, the system was designed for use on traditional terminals and the display format chosen reflects this. A simple association between field name and contents is used with markers delimiting the edge of the available data areas, as in the following example:

```
surname    [                ]
title      [    ]
firstname  [         ]
initials   [  ]
address1   [                        ]
address2   [                        ]
address3   [                        ]
address4   [                        ]
```

2.3. User interaction

An important consideration for a novice user is providing a consistent user interface, in the sense that all user interactions take the same basic form.

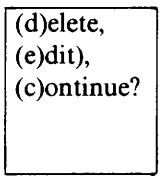
This has been achieved in this system by the use of a menu driver module which, given a definition of a menu, presents it to the user, and examines the user's responses, adapting itself as required.

A *menu* consists of a number of commands, each uniquely identified by a single letter, and having an explanatory string associated with it. These may be displayed in a screen area of any size and will be modified to fit as necessary. Input is by typing a single character, and, at present, no correction or confirmation is allowed.

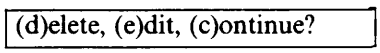
Thus, a menu such as the following:

identifier	name	explanatory string
d	delete	deletes current entry
e	edit	modifies current entry
c	continue	move on to next entry

might be displayed in a wide, deep window as:



or, in a one-line window as:



In either case, the response would be a single character as indicated, or ? requesting that the long explanatory strings be given.

All menus throughout the system are of this form.

2.4. Non-interactive interface

Whilst the majority of retrieval tasks can be handled through an interactive system, there are often tasks for which a non-interactive interface is desirable. It may be necessary, for instance, to extract statistics from a database on a regular basis for human consumption, or for presentation in a form-letter manner to a text formatter. Such tasks are more naturally handled by, for example, a shell script, and a non-interactive interface is thus desirable.

The non-interactive interface provided in this system allows qualifiers to follow the system invocation on the command line. Thus:

```
dbupdate -i admin people 'age>=60'
```

retrieves all records from the *people* relation of database *admin* who reach retirement age within the next 5 years.

The selected records are written to the standard output according to the following grammar:

record → **item** { ; **item** } \n
item → fieldname=contents

The contents of the field are written in the correct human-readable form for their type, e.g. times are written *hh:mm:ss*. Note that it is a restriction of the system that ';' may not appear in either fieldname or contents.

A library package exists for parsing input records of the above form, making their individual fields available to a user program.

3. SYSTEM IMPLEMENTATION

As mentioned earlier, the initial implementation was speeded up by the use of a number of facilities already present on our UNIX system. In particular, extensive use has been made of the INGRES database management system[2] for handling the data, and of the *curses* screen handling package [1] for easing the task of writing the user interface. This section describes the uses made of these systems and their interfaces to the rest of the system.

3.1. Database management

In the early stages, responsibility for management of the database was devolved to an existing database management system; in our case, this was INGRES.

The normal interface to INGRES is through a simple conversational monitor which, although powerful, is somewhat terse and unsuitable for a beginner. There is, in addition, a facility for interfacing C programs directly to INGRES. This facility, known as EQUEL, has been used in this project.

INGRES only handles 3 types of record element, namely strings (each of a prescribed maximum length), integers (of 1, 2 or 4 bytes) and reals (of 4 or 8 bytes). All of the field types described

earlier had to be mapped onto INGRES types.

At present, no information is recorded about the fields in a data record additional to that maintained by INGRES. This information amounts to the type of the object and its size in bytes. From this, it is necessary to deduce the internal associated type. This presents no problems for strings, integers and reals, but the decision to introduce dates and times into the system necessitated some additional structure to identify the different types.

All dates and times are represented in INGRES as 4 bytes integers in the normal way with dates being taken relative to January 1, 1970 00:00 GMT, and times relative to midnight on that day. These types are distinguished from integers and from themselves by the first two characters of the field name. Thus, a field representing a date and called *interview* would be represented as an INGRES domain called *_dinterview*. This mapping is invisible to the user. Likewise, times and combined date/times are handled in a similar way.

An early problem encountered with using INGRES concerned multiple updates. It is possible to request updates to all records satisfying certain criteria. This is implemented by retrieving all suitable records from INGRES and displaying them in turn. Each may now be edited if required. However, since updated records cannot be returned to INGRES until the completion of the retrieve, the edited records are retained locally and returned at the completion of the update. This necessitated the inclusion in each record of a serial number to identify each record and to ensure that it can be updated properly.

3.2. Screen handling

The primary capability required of the screen handling package is to display multiple windows on a screen in a relatively straightforward manner. The current version of the system uses three fixed size windows to display forms for editing, menu prompts, and a one-line status report. The ability to change these windows independently is essential.

The *curses* package of Ken Arnold was used to perform the screen manipulation. This provides the necessary facilities for handling a wide range of moderately dumb terminals, allowing the user program to treat the screen as a two-dimensional array of characters without the necessity to consider the differing characteristics of terminals.

In considering editing of forms, there was a great temptation to use an editor and avoid the problems. However, it was felt best not to require teaching a particular editor to a novice user and, instead, to present a simple on-screen editing facility. The facilities provided are limited and fairly crude: delete a character, delete an entry, move on to next entry and return to top of form. Whilst not allowing the sophistication of character insertion or easy movement around the form, these facilities were felt to provide the minimum necessary without imposing any great learning problem on the user.

As has already been mentioned, the basic interface to the user is through a series of menus. When called upon to print a menu in a particular window, the menu manager sizes both the menu and the window and decides how to format the menu. The options, in decreasing order, are:

one option per line, with one word identification
single line, with one word identification
single line, with single character identification

A limited help facility is available allowing a one-line explanation of each option to be displayed at some convenient place in the window.

3.3. Non-interactive interface

The format presented by the non-interactive interface has been described earlier. The breadth of queries supported by this interface is, in fact, greater than that of the interactive mode as it is far easier to specify complicated conditions in a textual form as opposed to a form oriented manner. This interface is, therefore, ideally suited for complicated processing of data as may be required to extract statistics for later human consumption.

A library package provides facilities for user programs to parse records arriving on the standard input and search for specified fields. This package has been used to facilitate label printing and form letter generation as well as generating complicated *troff* input based on the contents of a database. Other postprocessors are easily written as necessary.

4. CONCLUSIONS AND FUTURE WORK

The system was evolved to meet a specific need; namely, that of handling undergraduate admissions. It has developed into a more general system and is currently in use for maintenance of a number of information bases.

The design of the system is such as to ease modification to suit different environments. The menu-driven user interface can easily be removed and replaced by an interface using other input devices. With the imminent acquisition of workstations equipped with a number of differing input devices, further work will investigate the suitability of mice, touch screens, voice inputs and other devices in such a system.

Acknowledgements

The author would like to thank Mrs. Anne Aust for her consistent good humour despite being presented with a new version of this system nearly every day! Her feedback has helped greatly in the development of the system, especially in the design of the user interface.

References

1. K. C. R. C. Arnold, "Screen updating and cursor movement optimisation: a library package," *Berkeley UNIX manual, BSD2.8*, ().
2. M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES," *ACM Trans. Database Systems* **2**(3)(September 1976).

Automatic Generation of Syntax Directed Screen Editors

Theo de Ridder
Gert Florijn

HIO department IHBO "de Maere"
postbox 1075
7500 BB Enschede
Holland

(ridder@im60.UUCP)

ABSTRACT

From a new and effective automatic error-recovery scheme for LALR(1)-parsers the LYSE program generator is developed. LYSE produces for any language specified with LEX and YACC a syntax directed screen editor, in which editing and parsing are fully synchronized and integrated.
The editing functions are orthogonal, language and terminal independent, and require a minimal number of keys and keystrokes.
The semantics to realize an execution mode can be added to LEX and YACC in the usual way without restrictions.

1. INTRODUCTION.

Parsers, or more generally programs transforming grammatical structures, should not be made by hand anymore but generated with compiler-compilers. In the UNIX context LEX [1] and YACC [2] are available to produce lexical analysers for regular expressions and parsers for LALR(1) grammars respectively. Languages like awk, make and LEX itself are implemented with these tools.

Important drawbacks of the LEX+YACC toolbox are the primitive meta-syntaxyntax, the very inadequate and complex error handling strategy, and the batch nature of the generated compilers. However, on each of these points substantial improvements are possible within the same framework.

Extension of the meta-syntaxyntax can be done with preprocessors. In this way it is possible to create an attribute grammar system [3]. Another example is a preprocessor we made, using YACC, to tranform the standard COBOL meta-syntax into the YACC meta-syntax.

Our work on an improved error-recovery scheme will be published elsewhere. We found an elegant and effective solution for automatic error recovery that appeared to be symmetric for top-down LL(1) and bottom-up LALR(1) based parser generators.

Of course the capabilities of a 1-symbol-lookahead strategy remain restricted compared to multi-symbol-lookahead. However, the fact that 1-symbol-lookahead is very natural, simple and efficient at user and system level, is the base of our approach for LYSE.

There are some basic points that influence a parser essentially in an interactive context:

- *cursor control.* Complete synchronisation of the cursor and parser has the advantage that all text in front of the cursor is guaranteed to be correct at any moment.
- *error handling.* A strategy for automatic changes should be transformed in recovery advices.
- *redundancy insertion.* Automatic insertion of redundant tokens and completion of unique abbreviations will reduce the typing effort drastically.

- *visualized parse states.* It is helpful to visualize the current parse state with the symbolic representation of a grammar rule.

There is a sharp contrast between this view on interactive parsing and the approach of distributing ugly compiling processes over many windows in powerful bitmap-based workstations [4]. However, some windowing has to be done when immediate semantic evaluation is coupled to parsing. It is essential to separate source text screen editing and semantic input-output streams.

In the next section we will describe the way we integrated a screen editor interface. In section 3 some implementation aspects will be discussed. In the remaining sections some results and conclusions are given.

2. EDITOR INTERFACE.

In the design stage the first emphasis was on the synchronisation of cursor and parser movements, delaying decisions about the details of a screen editor interface. Validation experiments with an efficient cursor-parser coupling scheme revealed that a real ed superset was not feasible within an orthogonal approach. So we took the challenge to create a complete new screen editor interface based on the following global philosophy:

- *simple but complete.* For screen editors simplicity and completeness are not yet well defined concepts, but remain open to intuition, experience and subjectivity. We made an interpretation relative to existing screen editors [5, 6].
- *minimal keys and keystrokes.* There is a trade-off between the number of available keys and necessary keystrokes. We tried to minimize both with a small fixed set of special keys.
- *terminal independence.* The termcap database is used to establish a certain terminal independence. Modifications to a multi-window bitmap working station are left open.
- *language independence.* Automatic generation of a syntax directed screen editor for any language only from its LEX+YACC description does imply independence of semantic actions. Another consequence of the LEX+YACC role is the absence of abstract syntax manipulations, because these need extra specifications [7]. One should regard the produced editors basically as text screen editors with automatic help from the underlying syntactic structure.
- *orthogonal modes.* Our viewpoint in the single-or-multi-mode controverse is that multi-mode is acceptable as long as the modes are complete orthogonal.

Within those fundamental constraints there remain lot of possibilities for a concrete realisation. We made the following choices:

- *character usage.* Each printable character is directly echoed and, due to the left to right lexical analyser, always appended to the cursor. All function keys are represented with control characters.
- *granules.* A granule is the current piece of text, starting at the cursor, on which a cursor movement or editing function is defined. Except the granule <token> all others are grammar independent. Nesting and overlapping is possible. Specific granule selection is done with the corresponding granule key.
- *orthogonal key semantics.* The advantage of using granules are the orthogonality of the function key semantics and the small amount of required keys. For example there are only 4 cursor keys (<up>, <down>, <left>, <right>), and the effect of each changes with the current granule.
- *patterns.* A regular expression can be used as pattern, being one of the granules.
- *colors.* Another nice granule is color. For non-color terminals a 3-color system (<reverse>, <blink>, <underline>) is available.
- *escape mode.* Because each printable character is immediately appended as text during editing, a separate mode is necessary for typing things like patterns or filenames. This escape mode is extended as a meta-level with its own syntax and semantics, but editable in the same way as the other mode. In the meta-language any combination of edit functions can be composed as a new content of the <command> key. Additional features are automatic repetition within granules

and confirmation point indicators, enabling a facility as interactive global substitute.

- *execute mode.* There is an <execute> key that is just a switch to (de)activate semantic actions given in the YACC source.
- *history.* A history concept is applied in two different ways. Firstly to realise a sequential <undo> function and secondly to exclude the need for complicating meta-variables. The three meta-fields (<filename>, <pattern>, <command>) each has its own history that can be recalled and selected with a <menu> key.
- *screen layout.* The screen is divided in a text window, a current state information line, an escape mode line, and a window for error-messages, menues or semantic input-output streams.

3. IMPLEMENTATION ASPECTS

3.1. Introduction

Figure 1 presents a global overview of the LYSE-system. It shows that LEX and YACC are used to create the language dependent parser and scanner tables, which are combined during compilation with semantic actions and application independent interfaces.

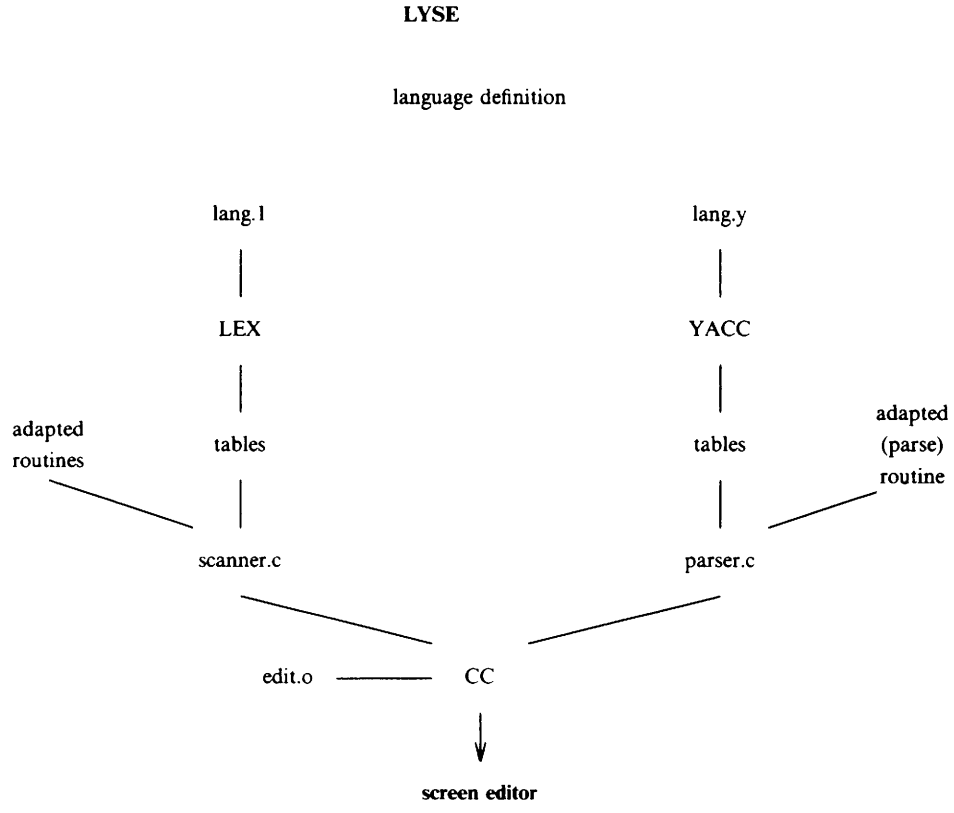


Figure 1. Global overview of LYSE

One of our basic design criteria was that existing tools should not be changed. This approach implied that changes could only affect the language independent routines and libraries. For a good understanding of our modifications we first give a brief and simplified description of the normal YACC and LEX parsing scheme.

Most language implementations contain several levels of abstraction, two of which are of interest here. There is the syntactical level, handled by the parser, where a given sequence of terminal symbols (tokens) is validated as part of the language involved. At the lexical level, the scanner groups characters into (terminal) symbols of the language.

YACC produces parsers for LALR(1)-grammars. The result consists of a set of tables (representing the grammar) and a standard routine ('yyparse') that operates on these tables. A stack is used to store the current state and the history of the parse-process leading to this state. Shift- and reduce actions are defined to extend or reduce this history.

LEX transforms a set of regular expressions with associated actions into a scanner. Routines match the expressions in the input-stream ('yylook') and execute the corresponding actions ('yylex').

Summarizing these definitions leads to the general overview presented in figure 2. It shows the various levels of abstraction in a language implementation, together with the routines that create these levels in a YACC-LEX based system.

Parsing Scheme

routine	functions
yyparse()	- process LALR(1)-grammar - recover from errors - execute actions
yylex()	- process regular expressions - return tokens (based on user actions)
yylook()	- process characters - return regular expressions
input()	- return characters from text

Figure. 2. Global parsing scheme.

3.2. Modifications for the interactive environment

The fundamental aspects of a syntax-directed screen editor impose new demands on this parsing scheme. Two of these aspects will be considered here: the synchronization of the parser (and scanner) with cursor-motions and the influence of the parser on the editing-process.

First however it is important to note that the edit-commands are executed at the lowest possible level, and that most of them are invisible for both the parser and the scanner. These only have to deal with possible side effects of the editing-process (e.g. cursor movements).

Synchronization of parser and cursor

One of the basic aspects of syntax-directed editing is the need for a complete synchronization between parser and cursor. All text in front of the cursor must be syntactically correct, while all text behind the cursor is invisible. Therefore the cursor position represents a kind of temporary 'end of file' marker which is moved around by the cursor movement functions.

It is obvious that forward cursor movements cause no problems: more input-text becomes visible, and scanning and parsing continue. Backward cursor motions, however, are more complex. The parser (and the scanner) must be brought back into the state in which they were, when this (new) cursor position was passed for the first time.

A straightforward solution would be to reset the parser and the scanner to their initial states, and to reparse and rescan upto the new cursor position. Unfortunately this would cause considerable overhead for large quantities of text.

Our approach is based on the fact that the current and the old parse state, that occurred when the cursor was at the new cursor position for the first time, are likely to have a (partly) common parse history. The synchronization after a backward cursor motion, can then be split up into a few tasks. First, the common part of the parse history must be identified, which implies that the common part of the parse stack must be found. After that the parser must be brought in the correct

state, by doing a partial reparse of the text.

```
stat   : ass_stat | if_stat;
ass_stat: ID '=' expr;
if_stat      : IF expr THEN stat ELSE stat FI;

(1) if a > 3
(2) then     x = 512
                 ^
(3) else     x = 1024
                 ^
(4) fi
```

Figure. 3. Example of synchronization.

Consider the small (incomplete) grammar and the sample program in figure 3. When the cursor is at the assignment-sign in line (2), as indicated by the ^, the parse stack looks something like this:

IF,expr,THEN,ID

After moving the cursor down to line (3), the stack-contents are:

IF,expr,THEN,stat,ELSE,ID

The common part of the two states is:

IF,expr,THEN

which implies that text must be reparsed, starting just behind the keyword 'then', and stopping at the new cursor position.

This mechanism is implemented by maintaining a separate storage of all tokens in front of the cursor. Every stored token has the depth of the parse-stack as it was just before the token was returned to the parser, associated with it. This enables a simple algorithm for finding the starting point of a reparse action. The token storage is maintained at the lexical level (i.e. in 'yylex'). The parser has been adapted to modify its stack-depth and state, when necessary.

This description is somewhat simplified, but it illustrates the basic principles. In reality, the resynchronization process is somewhat more complicated, due to multiple shift- and reduce actions on a single token. Furthermore, some information for the scanner must be saved (especially when LEX start-conditions are used).

Parser generated control

An interactive language environment should assist the user of the system, by sharing implicit knowledge. This should not only be done in the case of errors, but also in cases of redundancy.

Figure 4 shows how the adapted parse-routine for LYSE tries to meet these needs. It should be noted that error-messages are only given when repair is impossible. Of course, the cursor cannot be moved over a string that produces an error. It is placed in front of it.

Finding a redundant token, or a suggestion, implies searching various parse tables. However, finding a token is no guarantee that it really can be inserted into the text. This is caused by the fact that the parser uses numbers for token-representation, and therefore we need a mapping from token-numbers to textual representations.

- parsing error
  - only one token possible --> force insertion
  - more tokens are possible --> suggest one
  - refuse token --> error message
- parsing continuation
  - only one continuation --> append token
  - more than one continuation --> suggest one

Figure. 4. Parser-generated control.

YACC distinguishes two kinds of terminal symbols. First, the single character tokens, whose textual representation is in the YACC-input (e.g. '=' in figure 3). Their token-number corresponds to the ASCII-character concerned. Second, the more complex tokens, defined by the "%token" clause. This can be used for keywords, or things like identifier, and their textual translation is defined in the scanner.

We use a simple strategy to obtain the text representation for the latter class of tokens: the implementor must define it. In the YACC input-file, he is able to associate a translation string with a tokens:

```
%token FUNCTION /* %transl "function" */
%token ASSIGN /* %transl ":=" */
%token ARROW /* %transl "-->" */
```

Of course, only the tokens with a translation are candidates for insertion.

This scheme suggests a simple way to implement token completion. For every token read and processed by the parser, the matched text is compared with the translation, and (partially) substituted when necessary and possible. An adapted lexical definition of keywords is needed:

```
/* LEX-definitions for keyword-completion */

%%
func(t(i(o(n)?)?)?)?        return(FUNCTION);
whi(l(e)?)?                 return(WHILE);
```

3.3. Summary

Figure 5 outlines the parsing process as it used in the LYSE-generated screen editors. It should be noted, that the modfications make the use of LEX for obtaining a scanner more or less obligatory. Dura lex sed lex.

4. APPLICATIONS.

As a first application we generated a screen editor for a COBOL subset. In a COBOL program there is a lot of redundancy in the tokens. And indeed, as we expected, the amount of automatic inserts and token completion was impressing.

The next application was a data-entry system for a very small financial administration package written in awk. The awk language is characteristic for many other UNIX tools being very powerful for filtering sequential data, but quite unsuitable for interactive transformations. Another argument to use LYSE for data-entry was to show that the domain of the compiler-compiler technique is not restricted to compiler-writing itself. Only by specifying the syntax and semantics of a record in LEX+YACC we obtained an interactive screen-oriented interface with automatic error checking.

routine	functions
yyparse()	- process LALR(1)-grammar - reset parse stack - recover from/repair errors - execute actions - suggest tokens
yylex()	- token completion - add suggestions/insertions - process regular expressions - maintain token storage - return tokens (based on user actions)
yylook()	- process characters - return regular expressions
input()	- execute edit-commands - return characters

Figure. 5. Adapted parsing scheme.

As a more complicated and ambitious testcase of the LYSE philosophy the sh language was selected [8]. The benefits here are getting command history for free and creating the possibility for a lot of syntactic and semantic assistence. However, during the design and implementation some very serious prolems arose:

- *syntax.* It appeared not be an accident that there is not any precise syntactic description available. A number of complications made clear that sh is not designed or implemented in a way a compiler writer should do it. As an illustration of this statement the effects of the following sh statements on your own local system will suffice:

  ```
  for i in a=1 $a b=1; do echo $i; done
  <a >b >c cat <p >q <r
  <<'date'
  "'ls"
  ```

- *semantics.* The boundary between syntax and semantics in a sh program is so weak that it was a hard problem to incorparate an execution mode in the generated screen editor.

At the moment we are using LYSE to generate screen editors for other UNIX languages like the nroff ms-macro package, awk, C, and the Troll/USE query language [9].

5. CONCLUSIONS.

The first prototype of LYSE is implemented on a ULAB [10], a M68000-based system running UNIX V7. The compiled code for 85% of the complete editing interface took 55k bytes. In relation to its functionality this size is very small. But one has to realize the LEX and YACC tables will be large for complex languages. Running too many screen editors simultaneously in one machine will have the well-known performance degradation due to swapping. One of the solutions we have in mind is using a low-cost local area network for downloading generated screen editors in intelligent workstations.

Up to now we did not mention user-friendliness as an explicit goal. It is our opinion that high quality software based on fundamental knowledge will contribute automatically to a reduction of the complexity and chaos at the user level. Based on only 25 special keys, we realized an editing environment with a reference manual occupying just two pages (without hiding features!).

Finally it might be relevant to note that there are always a number of non-technical arguments for implementing a new piece of software. Some motives from our situation are:

- *education support.* Our main concern is computer science education on a practical level, and we want to offer the students an essential tool which was not available.
- *expertise exploitation.* We do have a long experience in teaching compiler writing and managing student projects in the field of compiler-compilers and error recovery strategies. It is a challenge to demonstrate the advantages of such knowledge for designing non-trivial but interesting applications.
- *product creation.* We don't claim to have made a real product yet, but want to show that our department is able to develop useful prototypes for the UNIX community. Our complete lack of adequate educational hardware facilities up to this moment might force us to negotiate about the availability of LYSE in stead of bringing it in the free domain.

References

1. Johnson, S.C., YACC: Yet Another Compiler-Compiler, Comp. Sci. Tech. Rep. No. 32, 1975, Bell Laboratories, Murray Hill.
2. Lesk, M.E., Schmidt, E., Lex - A Lexical Analyzer Generator, Comp. Sci. Tech. Rep. No. 39, 1975 Bell Laboratories, Murray Hill.
3. Katwijk, J. van, A Poor Man's Approach to Parsing Attribute Grammars, SIGPLAN Notices, Vol. 18, 10(Oct 1983), 12-15.
4. Thompson, C, Kelly, M, The teletype 5620 DMD: An Intelligent Graphics Terminal for UNIX, UNIX Review, dec/jan 1984.
5. Joy, W., An Introduction to Display Editing with Vi, Comp. Sci. Div., Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley.
6. Stallman, R.M., Emacs, the Extensible, Customizable, Self-Documenting Display Editor, Proc. ACM SIGPLAN SIGOA Symp. on Text Manipulation, Portland, Oregon, june 1981, 147-156.
7. Teitelbaum, T., Reps, T., The Cornell Program Synthesizer: A Syntax Directed Programming Environment, Comm. ACM, Vol. 24, 9(sept 81), 563-573.
8. Groen, J.H., Steenbergen, M.F. van, A Screen Oriented Shell Interface, IHBO "de Maere", HIO Dept., Enschede, 1984.
9. Wasserman, A.I., Kersten, M.L., A Relational Database Environment for Software Development, Rep. Nr. IR-86, 1983, Vrije Universiteit, Amsterdam.
10. Ulab System, Microproject Data Equipment, Haarlem.

CRS - a Powerful Primitive for Resource Sharing in UNIX

J. R. Nicol., G. S. Blair., and W. D. Shepherd.

Department of Computing
Lancaster University
Bailrigg, Lancaster
England

ABSTRACT

This paper focuses on a resource sharing system which we call 'CRS' (Connect Remote Shell). CRS is layered on top of the UNIX operating system and provides UNIX based local area networking environments with a powerful set of network services. The paper describes the use of the CRS system, followed by a detailed discussion concerning its design philosophy and the major aspects of its implementation.

1. INTRODUCTION

The domain of Distributed Systems is a diverse one which encapsulates several fields.

One of these is the field of Distributed Operating Systems (DOSs). There is no generally accepted definition of what a DOS is. We define a DOS to be system software which resides on top of a number of network nodes interconnected by a local area network. Further properties relating to the structure of this system software which we would insist on are the exhibition of a high degree of logical coupling and redundancy, full network transparency and unified resource access.

Since the requirements demanded of a DOS are presently so difficult to meet, no existing DOS can legitimately claim to have satisfied all (or even most) of them. The field remains a matter for further research.

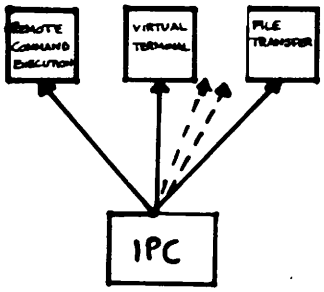
Another field of Distributed Systems concerns itself with the design of Resource Sharing Systems (RSSs). A RSS provides a mechanism which is specifically geared to permitting access to remote resources on the network.

The usual approach adopted in the design of RSSs is to design a set of high level protocols which are constructed on top of network dependent protocols. High level protocols provide high level services and support the sharing of these across the network. Each network service provided (eg file transfer, remote command execution, terminal connection) will be defined by a distinct protocol and will typically be accessed through the use of a correspondingly distinct primitive. Figure 3 illustrates this notion.

A less conventional approach to the design of RSSs is one which attempts to layer protocols on top of a single layer which supports general services. Higher layers can then be built on top of this layer in order to take advantage of the existence of the general services it provides. In this way, new and higher level network services can be provided. This approach (Figure 2) will invariably lead to a substantial saving of effort due to the avoidance of duplicated work. In contrast with the high level protocol approach another considerable advantage of the layering approach is that most of the network services provided can be made available via the use of a relatively small number of command primitives.

The advantages of the less conventional approach over the use of high level protocols led us to opt for the adoption of the former in the design of our CRS Resource Sharing System. The motivation behind our wish to design a Resource Sharing System becomes clearer when we consider some historical aspects of our former computing environment.

This environment consisted of a number of DEC PDP11/44 mini-computers running UNIX. In the recent past, each of these machines operated autonomously with each user's computing activities being centered around a particular machine. Circumstances changed when we obtained a



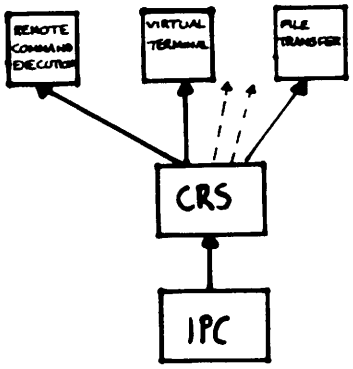


Figure 1. High level protocol approach Figure 2. CRS approach to resource sharing

Cambridge Ring local area network [1], since we were then presented with the possibility of sharing the available computing resources.

Our new computer network configuration demanded the provision of a set of network facilities. We therefore set out to develop an experimental type of resource sharing system which would be tailored to our kind of operating environment - one in which most users desiring to utilise the network would only occasionally wish to do so for performing small (but not necessarily trivial) inter-machine tasks.

Accordingly, it was important for us to provide such users with a simple but powerful interface to the network. It was our intention that the facilities available through the use of this interface should :

- allow the connection to any machine from any terminal
- allow execution of commands on remote machines
- be able to access resources (eg files) on remote machines
- be independent of any particular network configuration in use in a given environment.

In order to tailor our system to suit our computing environment, we aimed to compromise system features such as completeness, rigorousness and efficiency against those of practical utility and ease of implementation.

This paper consists of 5 main sections. In the remaining 4 sections, *section 2* outlines the CRS objectives in order to place the system in better perspective. CRS's only command primitive, the 'crs' primitive, is described and illustrated in *section 3. Section 4* describes CRS implementation issues in some detail and *section 5* presents some concluding remarks concerning our work on the system. Finally, a set of references for further reading is provided.

2. CRS OBJECTIVES

The design and development of the CRS system was essentially experimental in nature. Instead of setting out to develop a commercially viable end-product (and all that this would entail), we aimed to design a system which would be of particular use in our own and other similar environments.

Since CRS was intended to be experimental in nature, we did not envisage the system resulting in a marketable product. It is imperative for marketable systems to satisfy requirements such as system completeness, rigorousness and high reliability. We were consequently in a position to relax certain 'marketable' requirements with others of high importance, for example the need to install the system with minimal effort and the need to be highly usable by both regular and casual users.

Having stated our general objectives, we now present a more definitive list of major CRS system objectives (several of which were significantly influenced by the literature [2,3,4]). These were :

- to resist the temptation of modifying the UNIX shell, or the UNIX kernel, beyond a minimal level of necessity. It was our contention that satisfying the above constraint would eventually lead to a 'cleaner' design and implementation. Notice, that in order to provide a basic block protocol interface[5], we had to install a suitable driver within the UNIX kernel (just as would be required for any device being interfaced to UNIX).
- to provide a virtual terminal capability.
- to make possible the execution of commands on remote machines.
- to implement a design which would be flexible enough to permit a scheduling/load-balancing facility to be implemented, if desired, in the future. We felt that for our purposes, such a facility would not be required at present.
- to provide an integrated user interface to the system.
- to emulate the default security measures of UNIX

We first give an overview of the use of the system (section 3) then consider the implementation in some detail (section 4).

3. THE CRS PRIMITIVE

One of the major benefits of using the CRS system, is that all of the network facilities it offers are made available by the use of a single command - namely the 'crs' primitive. New users of CRS can therefore make comfortable use of the system after a short time.

In this section, we examine the primitive, demonstrating its power by means of example. The command syntax of the 'crs' primitive is shown below, in its most general form :

crs [-m] [-l loginname] [-s station] "command"

With the exception of the "command" string parameter, all other parameters are optional. Following the usual UNIX parameter handling conventions, the order in which the parameters appear in the command line is immaterial. The significance of the various 'crs' parameters is now discussed :

-m : if this parameter is supplied, the current message of the day ('motd') of the destination host will be supplied on the local hosts standard output channel. The 'motd' will not be supplied by default.

-l loginname : if this parameter option is supplied, an attempt will be made to log the local user into the destination host as the user with the -l specified loginname. Under such circumstances, the CRS system will request the appropriate password for that user (if any exists) - the password echo *is suppressed upon entry. By default, the local user is logged into the destination host as a user with very restricted rights of access.*

-s station : if this parameter is supplied, the user is specifying the name of the destination host which he wishes to log into. By default, the destination host would be taken as the local host.

"command" : this parameter must always be present. It can be almost any standard UNIX command that can normally be executed on the local host.

It is effectively the generality of the "command" string parameter and the ability of UNIX to redirect and pipe I/O, that leads to such an extraordinarily useful set of network facilities being made possible by the 'crs' primitive.

3.1. Examples Demonstrating The Use Of 'CRS'

For the sake of simplicity, we will assume that any passwords that are explicitly requested, in the following examples, are supplied accordingly and found to be valid by the system. A simple, though nevertheless useful example, introducing a possible use of 'crs' is :

crs -s 44R "who"

This command would display details of which users were currently logged into the destination host (ie 44R), on the user's terminal on the local host.

The use of 'crs' is demonstrated more generally, in the next few examples.

Example 1

crs -m -l fred -s 44M "sh -i"
44M - user fred - PASSWORD ? :

By executing an interactive shell command ('sh -i') on the destination host (44M in the example), the user would log into the destination host as 'fred'. Thereafter, the 44M's 'motd' will appear on the local user's terminal, followed by the familiar UNIX '$' prompt. The CRS user would have just created a virtual terminal, connected to the 44M host !. From now on, until the first ^D (ie EOF) is hit, the user can freely issue any further commands to the destination host just as if he was directly logged on to it, as user 'fred'.

Example 2

crs -s 44R "opr" < file1

This command would print out file 'file1' (belonging to the local user's current directory on the local host) on the destination host's printer. The command provides a primitive way of resource sharing - one printer can be shared between many machines.

Example 3

crs -l fred -s 44R "em thisfile"
44R - user fred - PASSWORD ? :

This command would allow the user to edit the remote file 'thisfile' (in 'fred's base directory on the destination host) from his current directory on the local host.

Example 4 (FILE-TRANSFERS)

crs -l jrn -s 44R "cat > project" < localfile
44R - user jrn - PASSWORD ? :

This command will transfer the file 'localfile', in the user's current directory on the local host, to the file 'project' in jrn's base directory on the destination host. The inverse operation can be achieved by the command :

crs -l jrn -s 44R "cat project" > localfile
44R - user jrn - PASSWORD ? :

which will transfer the file 'project' in jrn's base directory on the remote host to the file 'localfile' in the user's current directory on the local host.

From the above examples, it is possible to appreciate something of the 'crs' primitive's usefulness. Even so, it is capable of far more. An arbitrary example demonstrating the degree of flexibility offered by CRS, is that of a user executing a command on a remote host and then redirecting or piping its output to a program on some other host.

This kind of flexibility can clearly be of great advantage, but may demand some work on the user's part (in devising the appropriate command line for performing the task in hand). In this light, it is worthwhile stating explicitly that the limitations of CRS lie, to some extent, with the user's imagination.

4. IMPLEMENTATION ISSUES

We decided to adopt a layered software architecture approach in our design of the CRS system, ie we designed a number of layers of software which could themselves be layered on top of the UNIX operating system. We illustrate this notion in figure 1, below:

HIGHER LEVEL INTERFACES TO CRS
CONNECT REMOTE SHELL (CRS) LEVEL
REMOTE PROCEDURE CALL (RPC) LEVEL
BASIC BLOCK PROTOCOL (BBP) LEVEL
MINI-PACKET LEVEL

Figure 3 : CRS Software Layers

4.1. Lower Levels

At the Cambridge Ring level, mini-packets capable of containing 2 bytes of data circulate the ring. One level above this is the Basic Block Protocol (BBP) level [5]. The BBP is a 'best effort to deliver' message service which permits larger quantities of data to be sent than would normally be possible in a single mini-packet. It also has the advantage that a stream of mini-packets can be directed to a notional port†.

† A port is a protected queue of messages to which messages can be sent by any process knowing its name (station #/port #) and from which messages can be read by the one unique owner.

One level above the BBP level, is the Remote Procedure Call (RPC) level [6]. The RPC is a transaction based message passing facility, conforming to the model of distributing computing of a series of processes acting as clients and servers. The RPC and lower levels take care of all errors in the network. Above this level, the user does not need to know or worry about network details.

The services provided by the CRS system are then layered on top of the RPC protocol level. Above the CRS system level, higher level interfaces to CRS can be layered.

4.2. The CRS Layer Implementation

We now consider, in some detail, how the CRS layer is implemented.

On each host in the network, there exists a login server process which waits for service requests from some other host, ie for some network transaction to be set up between two hosts.

When a user enters a 'crs' command line, a client process is spawned. The client then communicates with the login server on the appropriate host (as indicated in the command line). If the login server determines the login message (sent to it by the client) to be invalid, it informs the client of the login failure and thereafter ignores the request. Otherwise, the client and login server processes engage in a handshake protocol, exchanging unique station/port pairs.

This addressing information is then passed to a number of new processes which have since been spawned by the client and server processes. Using this information, these new processes are able to set up formal communication channels (between themselves) across the network and, in this way, provide a mechanism for general inter-process communication (IPC). The original login server is then able to resume its wait to service other possible CRS session requests.

Schematically, the sequence of events which occur over time and the channels of communication between the various processes, are illustrated in figure 4.

Network channels are set up between the local client/input filter and output filter/local server pairs respectively. A number of pipes† † A pipe [7] is a UNIX mechanism for performing inter-process communication. are also established between CRS processes which exist on the same host. With reference to figure 4, we now give a relatively high-level description of the operation of each of the processes involved in a CRS execution.

The *local client process* reads its standard input from the physical terminal by default. It buffers data, transmitting the contents of the buffer either when a newline character is detected, or when the buffer becomes full. The buffered data is transmitted across the network to a destination host process which we refer to as the 'input filter'.

The *remote input filter process* obtains its standard input by reading data from the same port to which the local client is sending its standard output. Any data read by the input filter is then piped to the remote command process.

The *remote command process* is responsible for performing the task received from the network. It receives its input from the input filter and any output produced will be piped to the output filter. Notice that the remote command's output and error diagnostics are piped to the output filter process via distinct pipes.

The operations of the *output filter* and *local server processes* are effectively symmetrical to those of the local client and input filter processes respectively. One important distinction between the operation of the input and output filters is that the latter operates by polling two different pipes. One pipe is for the remote command's standard output channel and the other is for its standard error channel. Whenever the output filter detects any data in the pipe it is currently polling, it will empty that pipe (transmitting the data back across the network to the local server) before it recommences polling - starting with the other pipe. This mechanism is used to synchronize data arriving from the standard output and error channels of the remote command, at the other end of the network.

We devised a simple protocol which enables the local server to determine on which channel it ought to direct the data it has received (ie the standard output or error diagnostic channel) from the output filter. Hence, we were able to preserve many of the channel handling characteristics of UNIX.

4.3. Ending A CRS Session

From the above, we can see that during a transaction, a chain of processes are set up in such a way that they are able to communicate with one another around the network. As soon as a particular CRS session completes, it is clearly necessary to terminate each of the processes in the chain.

We designed a 'tidy-up' protocol which ensures that whenever any process in the chain exits, this in turn, causes all other processes (in the chain) to exit. The exact form of this protocol will depend on the event which occurred to signify the end of the session.

The most general form of the protocol is employed when the first process in the chain detects an end-of-file character in the local end's input character stream (eg when a user hits ^D whilst using the virtual terminal facility, or when a file has been read to exhaustion).

The effect that this will have, is to spark off a chain reaction around the CRS session's processes. Each process in the chain stimulates the next (by providing the information that an EOF character has been detected at the local end) prior to terminating itself. This effect is illustrated conceptually, in figure 5.

In order to accelerate the rate at which this terminating chain-reaction process occurs, we fully exploited the fact that a breakage in a UNIX pipe can be detected. Hence, for example, when the input filter dies, the remote command process will know to terminate when it detects the resulting breakage in the pipe which used to 'connect' these two processes.

A modification to the above protocol is required to handle the second reason for termination. This is when the remote command completes its task. The command process exits resulting in the input and output filter processes detecting breakages in their respective pipes. As in the previous case, the output filter is responsible for instructing the local server process to exit, before exiting itself. The major difference is that this time, **it is the responsibility of the local server process** for ensuring that the local client process knows to terminate.

We found this protocol to be a both reliable and efficient way of 'tidying- up', at the completion of a CRS session.

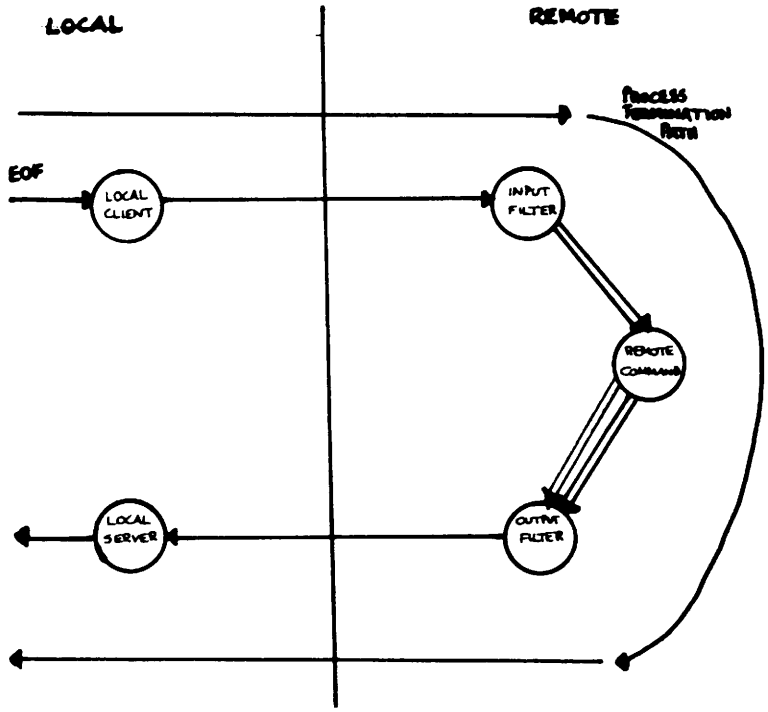


Figure 5. Chain Reaction effect of Process Termination

4.4. Orphan Handling

One major problem which manifests itself in the domain of distributed computing systems is the problem of how to handle orphan processes when they occur.

An orphan is generally a server process which is left stranded on a remote host machine when, for example, the client processes (with which the server is trying to communicate) die for some unexpected reason - such as the host machine crashing. Since it is impossible to ensure that server processes immediately become aware of client process crashes, they can potentially become orphans, which continue to work as before. Orphans unnecessarily consume processor time on the remote host and can, perhaps, even interfere with the operation of other processes involved in future transactions. Hence, orphan processes are clearly an undesirable phenomenon.

Lampson [8] has devised some complicated protocols for combating the problem of orphan processes. Despite the rigorousness of these protocols, we decided that they were unsuitable for our purposes, since they would significantly add to the CRS system's implementation complexity, and, their adoption would also place an unacceptably high performance overhead on the system. Both of these consequences conflict with our previously stated system objectives.

As a trade-off measure, we designed a simple, but effective, light-weight protocol which we believe to be a more satisfactory mechanism for handling potential CRS system orphan processes. The protocol is illustrated in figure 6.

Essentially, the remote end checks up periodically to verify whether the local end is still active. More specifically, the input filter process will timeout if it has not received any input within a given time period. Under such circumstances, the input filter interrupts the local server process, requesting it to respond to the timeout. Provided a suitable response arrives from the local server process, the input filter is safe to assume that the local end is in an acceptable state. The CRS system can therefore continue its operation as before.

If however, such a timeout occurs and no suitable response is received from the local server process, the input filter assumes that the local end of the system has crashed. Consequently, the input filter informs the remote command and output filter processes of the local end's assumed state, whereupon each of these (now orphan) processes will exit.

Our use of this protocol enables us to guarantee that no remote orphan process can ever

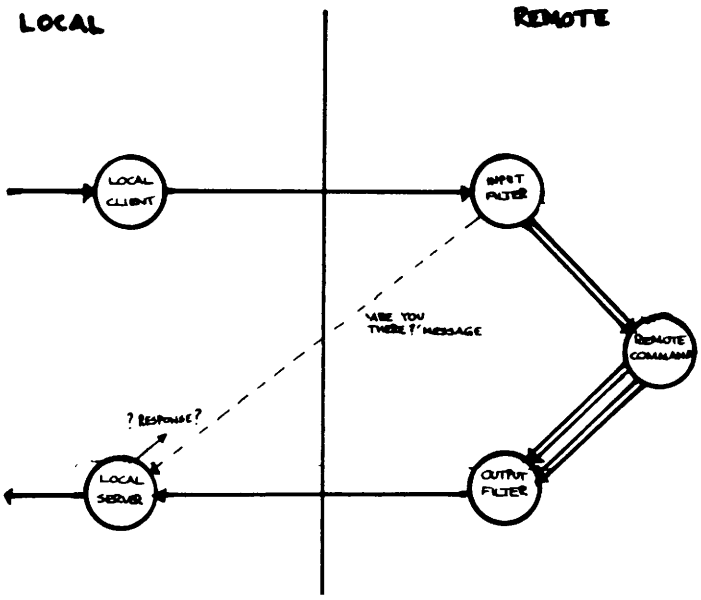


Figure 6. Orphan Handling Protocol

survive for longer than the duration of the input filter timeout period.

4.5. Higher Level Interfaces

It is our intention that higher level interface commands can be layered on top of the 'crs' primitive. The motivation behind this intention is to permit the more casual system users, who might only require to very occasionally make use of one or two specific CRS facilities, to do so without having to understand or even know about the underlying CRS details.

An example of one such higher level command interface (which has been implemented), is the 'remlogin' command. When the user wishes to log into another host, he simply enters 'remlogin' at his teletype, which then presents him with a UNIX-like login interface. Remlogin will extract the required login details from the user and then map these onto an appropriate 'crs' primitive invocation.

Of course, the main advantage of 'remlogin' is that it hides the 'crs' invocation mechanism etc from the user.

4.6. CRS Protection And Security

On the note of security, we previously mentioned that it was our aim to design the CRS system in such a way that it would be able to provide at least as powerful security and protection measures as UNIX. This objective has been fully achieved.

Moreover, passwords cannot be picked up by such means as 'wire-tapping', since passwords traverse the network in encrypted form only.

5. CONCLUSION

In the light that all CRS system objectives have been satisfied, we consider that our work on the project was successful.

We were particularly pleased with our incorporation of all system network services into a single command primitive, since, few other systems have succeeded in their bid to achieve such an extremely integrated user interface. Another notable aspect of CRS, is the high degree of flexibility offered to its users for remote command execution around the network. The vast majority of UNIX

commands and mechanisms can be handled by CRS, in a way which as previously stated, is limited to some extent by the user's imagination.

As regards system performance, there is some overhead involved in setting up the initial connection, therefore single commands can be slow. However, once the initial connection has been set up, the data transfer rate of CRS across the network, is roughly comparable with the rate at which UNIX is able to 'cat' data locally. The response time of CRS, in its interactive mode, is also most acceptable - on this basis, few CRS users are aware that they are working in a network supported resource sharing environment.

The CRS system has now been developed to a standard whereby it can be installed in any UNIX based local networking environment, with minimal effort and, yet, it is capable of providing a real and practical set of network resources to a network resource sharing environment.

The system as it currently stands, however, is neither rigorous enough, nor foolproof enough to be considered as a commercially viable product. Since this was never our intention from the outset of the project, we were able to use this situation to our advantage. In particular, we were able to compromise certain system features in order to tailor CRS to the kind of environment in which we intended it to be used.

Looking to the future, we believe that research interest in the design of fully distributed operating systems (as defined in [9]) will continue to increase as experience in the field accumulates. Nevertheless, systems such as CRS still have a vital role to play in bridging the gap between a completely unconnected set of operating systems and a fully distributed operating system.

References

1. Wilkes, M. V., Wheeler, D. J., "The Cambridge Digital Communication Ring", Local Area Comms. Network Symp., Mitre Corporation and National Bureau Of Standards, Boston, May, 1979.
2. Hwang, K., Wah, B. W., Briggs, F. A., "Engineering Computer Network (ECN) : A Hardwired Network Of UNIX Computer Systems", Proc. National Computer Conference, Vol 50, AFIPS Press, May 1981, pp191---201.
3. Nowitz, D. A., Lesk, M. E., "Implementation of a Dial-Up Network of UNIX systems", COMPCON Fall '80, 21st IEEE Comp. Sci. Conference, pp483-486.
4. Antonelli, C. J., et al. "SDS/NET - An Interactive Distributed Operating System", COMPCON Fall '80, 21st IEEE Comp. Sci. Conference, pp487-493
5. Blair, G. S., Mariani, J. A., Shepherd, W. D., "A Practical Extension to UNIX for Interprocess Communication", Software Practise And Experience, Vol 13, pp45-58, 1983.
6. Nelson, B. J., "Remote Procedure Calls", Internal Report, CMU-CS-81-119, Dept Of Computer Science, Carnegie-Mellon University (1981), Ph.D.
7. Ritchie, D. M., Thompson, K. L., "The UNIX Time-sharing System", CACM, 17 (7), pp365-375, 1974.
8. Lampson, B. W., "Applications And Protocols", in Distributed Systems - Architecture and Implementation : An Advanced Course, Springer-Verlag, 1981.
9. Blair, G. S., "Distributed Operating System Structures for Local Area Network Based Systems", University Of Strathclyde, Ph.D. Thesis, December 1983.

Honey Danber - The UUCP of the Future

Peter Honeyman

Princeton University
Princeton, New Jersey 08544

Dave Nowitz

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Brian E. Redman

Bell Communications Research
Whippany, New Jersey 07981

(ber@yquem.UUCP)

ABSTRACT

The UNIX to UNIX Copy Program (*UUCP*) embodies many good ideas for an inexpensive file transfer and remote execution network; however, the current implementation is over five years old and is troubled by many problems.

This paper describes a new implementation. The main goals were to make *uucp* more robust, secure, powerful, and maintainable. The major activities were:

- A massive code reduction. Four core programs comprise the system: *uucp, uucico, uux,* and *uuxqt.* Code that did not directly implement the purpose of these programs was excised.
- The connection algorithm was rewritten to persevere in the face of adversity and to provide a mechanism that enables the administrator to incorporate new calling devices easily.
- The spooling mechanism was replaced with one that hashes the queued files by remote system name.
- The USERFILE syntax was discarded and replaced by an extremely flexible and intelligible mechanism with practical (secure) defaults.
- The code was reconciled to meet the authors' standards for programming style, robustness, and maintainability.

1. INTRODUCTION

We describe a new implementation of UUCP which solves most of its problems and provides many useful enhancements. The current standard version is described in [1-3]. This paper discusses many of the significant changes.

2. SITE NAME HASHING

Very large directories take significant time to search. When */usr/spool/uucp* becomes very large UUCP performance degrades badly. In order to keep the spool directory size in hand, separate subdirectories are now used. Data files representing a particular remote system are placed in a directory named with the remote system. The UUCP programs which access these files change their working directory into the directory associated with the remote system and perform their operations there. Besides improving performance on heavily trafficked sites, this conveniently isolates the data. Backlogs for some sites will not affect communications with others. This

partitioning has been extended to include separate directories for administration, sequence files, and logging. The latter two directories contain files for each active remote system.

2.1. Side Effects

Uuxqt commands now execute in parallel, one for each remote site. The maximum number of simultaneous **uuxqt** commands may be controlled by the contents of a file (*/usr/lib/uucp/Maxuuxqts*).

The name of a site no longer must be encoded into a file name. The name of the directory denotes the site. Since the directory name is the limiting factor, site names of up to fourteen characters are permitted. We have tried to isolate this limitation to a single manifest constant. Thus systems which support longer file names may allow greater site names. Perhaps future versions will use the directory name as a pointer to the real site name. Even conservative use of the character set would allow over 64^{14} different names.

3. SECURITY

We have completely revised the syntax and semantics of the **USERFILE**. It's new reification is called **Permissions**. */usr/lib/uucp/Permissions* provides increased functionality and flexibility. It contains statements which customize the behavior of UUCP for any or all systems by identifying them with the login name they use or their site name. The statements described are used in conjunction with two basic types of entries which are

LOGNAME=

for remotely initiated connections and

MACHINE=

for locally initiated connections. There must be a LOGNAME= statement for each login which might be used by a remote system. The minimal contents of the **Permissions** file is:

LOGNAME=uucp

This specifies that a remote system may login as *uucp* and will be restricted by the defaults.

The permissions fall into three basic categories:

File system access

Command execution

Identity verification

File system access permissions dictate which directories a remote system may access for reading, and separately which may be accessed for writing. This is done by specifying any combination of the keywords **READ**, **WRITE**, **NOREAD**, and **NOWRITE** followed by a ':' separated list of directories. In the example:

READ=/usr/ber:/usr/honey:/usr/dan WRITE=/tmp NOREAD=/usr/ber/secret

a system can read any files which the UUCP login can access in the directories */usr/ber*, */usr/honey*, and */usr/dan* except those files in */usr/ber/secret*. It can also write into any file in */tmp*.

The command execution permissions specify commands which can be invoked.

COMMANDS=rnews:lpr:opr:rmail

shows that the site associated with this line can execute any of the listed commands and no others. If a path name is specified then it must match the command requested by the remote system exactly.

Identity verification is accomplished in a variety of ways. *CALLBACK=yes* is most restrictive. It requires that when the remote system calls, it must hang up and wait for the local system to return the call. This provides the greatest assurance that we are communicating with the correct party.

SENDFILES=no prevents the *master/slave* protocol from being reversed when the remote system calls. The result is that although the local site may have data queued for the remote site, it will not transmit it if the remote system originates the call. This is slightly less restrictive than *CALLBACK=yes*.

REQUEST=no indicates that the remote site may not request files from us under any circumstance. In order to receive files the remote user must have an agent on the local site to initiate the transfer. This restriction applies no matter who calls whom.

Finally, when *VALIDATE=* appears, the specified systems must have logged in using one of the associated logins indicated. For example:

```
LOGNAME=Umarx \
  VALIDATE=chico:harpo:zeppo
```

will cause UUCP to terminate the conversation if *chico*, *harpo* or *zeppo* log in with a name other than *Umarx*.

The default permissions are:

```
LOGNAME=uucp \
  WRITE=/usr/spool/uucppublic \
  REQUEST=no SENDFILES=no \
  COMMANDS=rmail:rnews
```

Note that the statement must appear as a single record. Lengthy statements may be continued with '\'. Appendix I is a sample **Permissions** file, Appendix II is the description emitted from a program which checks the **Permissions** file (described later).

These mechanisms allow a site to implement secure features where they are needed while still allowing considerable flexibility in the treatment of sites based on knowledge attributed to the login method and remote name. This concept allows UUCP to be useful both with tightly coupled well trusted machines and the loosely coupled less secure dialup network environment.

4. A RATIONAL CONNECTION FUNCTION

The third general area of revision is that of the connection function, **conn ( )**. The goals in revising conn( ) were to bolster its robustness, handle more types of connecting hardware and to provide a mechanism for easily utilizing new apparatus. This was done by splitting the hardware-dependent operations into two routines. One understands the needs of *callers* (e.g., how to interpret the fields in the **Systems** file, nee *L.sys*). The other understands how to manipulate the hardware to connect to a remote resource (parochially known as dialing). Conn( ) uses a table containing the name of each caller (ACU, Micom, TCP, Sytek, etc.) and a pointer to the function which manipulates it. The caller function uses a table of *dialers* which contains their names (212, Penril, Vadic, etc.), and pointers to the functions which perform dialing. Not all callers have associated dialers. Another way to describe it is that the caller specifies the nature of the network and the dialer dictates the mechanism to access such a network. Thus we find currently that there are many dialers for the ACU caller because there is a variety of hardware to access the Direct Distance Dialing network. But we have only one interface to the DATAKIT network so the calling routine for it may as well (and indeed does) include the dialing function. If alternate interfaces become available then the dialing function will be split out and the caller will be able to choose among them. The enhancement procedure for hardware unknown to us at the time of this writing then entails supplying a new caller or dialer function, updating the tables, and recompiling.

Given these tables which contain identifiers, we provide a new syntax for the **Devices** file (formerly called **L-devices**) in which the caller hardware and dialer function are specified. Appendix III is a sample **Devices** file, Appendix IV is a sample **Systems** file. (**L-dialcodes** has been renamed **Dialcodes** but is unchanged from previous versions.) Note the existence of a *chat* script in the **Devices** file. This is a syntactic convenience used to isolate the differences in various switches so their similarities can be exploited in a generic caller routine.

Conn( ) is called with the name of the remote system and behaves as follows:

```
for each entry in Systems which matches the argument
   for each caller in Devices which matches this entry
      if this caller doesn't use a dialer
         attempt to connect
         if successful
            return file descriptor
      else
         for each dialer which can be used with this caller
            attempt a connection using the dialer function
            if successful
               return file descriptor
try again later
```

This mechanism allows us to exploit all the alternative hardware at our disposal in an attempt to make a connection. The priorities of the various alternatives are expressed in the ordering of the entries in the **Systems** and **Devices** files. A subtlety of interest is the use of the *ANY* keyword in the *class* fields of the **Devices** file. This affords more flexibility in determining which connection hardware to use. For example one may wish to connect to some systems at a speed below the hardware maximum to counteract deficiencies in their terminal drivers.

5. RETURNING STATUS FROM REMOTELY EXECUTED COMMANDS

Specific knowledge of the commands invoked by **uuxqt** has been removed. All commands are handled in a uniform manner.

In previous versions the **mail** command was handled specially. For all commands executed other than **mail**, the status returned was unconditionally reported to the invoker. For **mail** however, no status was returned if it succeeded. But, if it failed, **uuxqt** would return the standard input as well as the status. This was justified by the fact that **uuxqt** was principally used for remote mailing. When netnews became widely used, it too was a logical candidate for exceptional handling within **uuxqt**. However, in order to deal with commands in a flexible and general manner, we provide three options to **uux** which modify **uuxqt**'s behavior. They are:

—n do not request error notification (overrides default)
—z request success notification (overrides default)
—b return standard input on failure

In all cases of a failure, the standard error is mailed to the originator. Appendix V is a sample message from UUCP resulting from a failed **rnews** command.

6. COPING WITH BAD DATA AND INADEQUATE ENVIRONMENTS

Although the new implementation of UUCP is laudably robust in the creation and transmission of data, it must still deal with improperly formatted files that are the result of lesser versions or system problems. When UUCP evaluates a command or execution file it checks that the contents are plausible. When corrupt files are identified, they are moved to a special directory and processing continues with the next file. A daemon checks this directory periodically and informs the administrator of its contents.

The new UUCP also checks the level of the file system using the **ustat** system call (an equivalent user level routine is provided for systems lacking **ustat**.) A special error code is transmitted to the remote system if there is insufficient space. The event is logged and the conversation is terminated. If UUCP receives this error indication from a system, it will disengage from the conversation. Scans of the log files will reveal these situations and the local administrator may wish to inform the remote counterpart.

7. SEQUENCE NUMBERS

The sequence number is seven hex digits. Four of these digits are derived from a sequence number file. The remaining three digits represent a cache of sub-job numbers which modify the base number. Thus a command will require only one access to the file for up to 16^3 sequence numbers. Since we use separate spool directories for each remote site, name clashes will not occur when different systems send us the same file names. The initial base number is selected randomly, reducing the possibility that we would generate names that clash on a remote site running an old version of UUCP. The use of seven hex digits and the fact that a different number sequence is maintained for each remote site allows us to avoid the use of alphabetic sequences which are aesthetically unpleasing and potentially offensive.

8. A NEW LOCKING MECHANISM

Previous versions of UUCP relied on the modification time of a lock file. It was assumed that if a file was older than some threshold that it was invalid. This led to problems when UUCP (or other programs sharing resources with UUCP) executed for extended periods of time. One awkward solution was to have programs periodically touch the lock file. Our solution relies on a property of System V which enables a process to determine if it can kill another without actually disturbing it. The identifier of the process using a resource is recorded in the lock file. When another process examines the lock file it issues a **kill**(0) to the process id contained within. If the **kill** succeeds this indicates that the specified process is still running. If it fails the lock file is assumed to be invalid. The code to add this function to other versions of UNIX is slight and straightforward. Alternatively a routine could be written which uses **ps** to glean the same information, though this would be considerably less efficient. For systems that don't implement this property of **kill**, UUCP may be configured to use the old mechanism.

9. PRIORITIES

Versions of UUCP since the second have had the capability to grade file transfers by use of the "—g" option. The option had almost no effect in practice. In our version, the grade option does something useful. When command files are gathered up, they are sorted, so the grade serves to order the processing of command files. When used with **uux**, the grade option is bound to the execution file as well. Thus upon receiving execution files, they are similarly gathered and sorted and the grade again dictates their execution order. The grade option could be used, for example, to provide mail with a higher priority than netnews.

10. OTHER CHANGES

Uucp can now copy users' files which the UUCP login cannot read by making a copy in the spool directory using the user's access privileges.

Log files are more informative and less chatty.

We have added the 'e' protocol for error-free links where the packet size is the file size.

An exponential backoff with a twenty-three hour maximum is used rather than a constant period for retries. A connection will be attempted forever at the maximum period. This may be overridded by appending a semicolon and an integer to the *when* field of the **Systems** files.

11. FORWARDING

Users have always been confused by the lack of a forwarding mechanism in UUCP. Typically they assume that the syntax used by **mail** (i.e., mail ucbvax!bellcore!psl) is correct for **uucp** (uucp file ucbvax!bellcore!~/psl/file). Until very recently **uucp** could not handle such a request. In response to this Mark Horton wrote a command (**uusend**) which was distributed with 4.1bsd. The **uusend** command accepted the mail-like syntax and used the identical mechanism. The way it worked was that a **uux** command was issued to execute **uusend** on the next site in the path. There the **uusend** command issued another **uux** for the next site until the destination was reached. The catch was that the **uusend** command must be permitted to be remotely executed on each intermediate site. In the

version of UUCP that we used as a base a forwarding mechanism was incorporated into the code. Unfortunately it appeared as a special case requiring additional semantics for **uuxqt** on the remote system. This code was immediately excised (over 500 lines).

The **uusend** model was correct. However **uucp** now generates the appropriate **uux** command . * In fact **uucp** is merely a special case of **uux** where the remotely executed command is **cp**. Perhaps it could be replaced with a one-line shell script? This is better than the former forwarding scheme because it is cleaner and it will work with any version. The caveat remains that the remote systems must allow the **uucp** command to be executed by **uuxqt**.

12. NEW CAPABILITIES

There are two additional options used with the **Permissions** file which have the potential to allow users of UUCP to deal with some unresolved networking problems. They are the *MYNAME* and *PUBDIR* options.

As described previously, the **Permission** file allows the system to behave differently depending upon the remote site with which it is in communication. The *MYNAME=newname* option instructs UUCP to behave as if the local system were named *newname*. *PUBDIR=/newdir* causes the local system to use *newdir* rather than the standard spool directory for the remote system.

One can set up a gateway machine for a group of others. By instructing remote systems to connect to the same physical machine using different logins, data destined to the satellites will be transferred unwittingly to the gateway. The data accumulated in the special spool directories for the satellites can then be transferred to them and their UUCP systems can act upon it as if it had come directly from the originating system. Similarly data destined for a remote system from one of the satellites can be redirected, using the *PUBDIR* option, to a special directory which is then transferred to the gateway for actual transmission to the remote system. This scheme is well confined and invisible to the users.

Another application might be to provide increased bandwidth among machines. Let's say machines *tempus* and *fugit* have a great deal of traffic between them and their communications link is not sufficient to dispose of all the data. Then *tempus* can think of *fugit* as two machines, *fugit1* and *fugit2*. Likewise *fugit* will know of *tempus1* and *tempus2*. Then it is a simple matter for traffic between these two machines to utilize two separate routes. UUCP will have two simultaneous non-interfering connections between *tempus* and *fugit*, thereby doubling the traffic capacity. A similar use would be to have high priority traffic sent to a machine using one name and other data sent using another.

13. CONFIGURATION

Our new version has been described as a tribute to the C preprocessor. It has to run on a variety of UNIX versions (including hybrids). So as well as containing *ifdefs* for **V7**, **V8**, **BSD4_2**, and **ATTSV**, it also allows for inclusion of various features such as **ustat**, an improved **kill**, and a high resolution sleep.

It has to display certain functionality optionally so that local installers can chose the behavior which best suits their environment. For instance the emission of sensitive information while running with debugging turned on can be permitted, prevented or tailored by use of a macro which checks that the group id falls within a specified range.

It is necessary to be able to configure it to interface with various user programs. For example, even though lock files don't belong in UUCP's spool directory, the other programs that care about UUCP's lock files might not all be changed. We provide the option of compatibility for the lazy.

Parameters likely to be changed out of necessity or preference have been gathered in one place (**parms.h**) for easy identification and manipulation.

14. INSTALLATION

Although this version of UUCP has been designed to fall back on sensible defaults where some files are not present (such as **Maxuuqts**), other files and directories are too critical to do without (e.g., **Systems**) and for many, the modes are crucial. Also, since this version is significantly different, old working data must be converted to conform to the required formats. A comprehensive *makefile* provides the basis for installation (we thank those who were then 6.0 developers). A conversion shell script is provided as well as a program to set up the required files and directories and set their modes correctly. An additional program (**uucheck**) verifies the procedure. It can also interpret the contents of the **Permissions** file to the user in very descriptive terms (i.e., system so-and-so can do such-and-such and is restricted from this-and-that). See Appendix II. Ideally a recipient of the new system will be able to edit **parms.h**, and type "make install" to generate a working system.

15. MAINTENANCE

Due to the anarchic and volatile nature of the largest network which UUCP supports, not all requests which are generated are completed. For one reason or another data files will become orphaned, command and execution files widowed, temporary files will strive for immortality, core files will be born (not by UUCP of course) and *dead.letters* will litter directories. Only a program can expend the time and effort required to manage a busy system routinely.

One of the first programs we threw away was **uuclean**. This left us without an automated mechanism to sweep all of UUCP's litter under the rug. Over a period of several weeks we noted our actions and our thoughts as we manually handled the situations described above in the most considerate manner. From these observations a set of heuristics was compiled which effectively deal with the deficiencies of an imperfect system. They are represented by a program (**uucleanup**) that deals handily with the typical data (mail and netnews). Unanticipated problems are dealt with inelegantly in the style of **uuclean**, but are recorded to facilitate the incorporation of additional knowledge as it becomes necessary. Data is not arbitrarily destroyed. Rather, considerable effort is expended to see that it is returned to the originator (or sent on to the destination if enough information is available). Users will no longer be frustrated by a cryptic message like

> D.grouchoN1234 deleted after 7 days.
> Could not contact remote.

We also provide programs to nudge **uucico** into action and to aid in debugging connections. New **uustat** and **uulog** commands help users and administrators monitor activity. The *uudemons* have been functionally separated into a polling daemon (which uses a syntactically pleasant file for schedules), an administrative watchdog, and a janitorial daemon.

16. DOCUMENTATION

Real programmers don't need auxiliary documentation, they read the source code. In deference to those programmers we have put considerable effort into making the source code readable. But the UUCP user population will include many more who don't want to read the code (their loss). For them we have revised the manual pages to more accurately reflect the system. Many new manual pages are included covering the utilities mentioned. Even manual pages for the embedded programs **uucico** and **uuxqt** have been written. Other documents will be published [4] or are in preparation now by the authors. A comprehensive administrative guide will be included with the *Official* release.

17. FUTURE WORK

The wishlist grows continually. It includes:

More tuning.

Encryption of data files.

Non-spooling for high speed networks.

Independent debugging of different functions.

Better error recovery resulting in fewer fatal errors.

Compile-time options to give the installer more control over space/time tradeoffs.

Number translation for different hardware which will permit more generic dialer functions.

18. CONCLUSION

Uucp is now in its third major version. The new system is healthier and more versatile than its predecessors and the code is easier to work with. We have attempted to provide a system that is useful for all sorts of networking applications and one which can be painlessly enhanced to accommodate new hardware. We hope that this version will be used to bring UUCP maintainers back to a common system so that their future developments may be shared more vigorously.

Acknowledgements

We thank the following for their substantive work and comments: adiron!bob, axiom!smk, cbosgd!mark, masscomp!trb, ihnp4!gjm, ittvax!swatt, mouton!karn, ncsu!mcm, rti!trt, sun!shannon, watmath!arwhite, icarus!alb, watmath!dmmartindale, whuxlb!eric. For moral support, thanks to decvax!larry, exodus!dvw, gummo!mmp, parsec!kolstad, rabbit!ark, research!dmr, research!doug, utzoo!henry, vax135!martin, watmath!bstempleton.

Thanks to allegra!jpl who reviewed the code with us and contributed some of his own. Our appreciation goes to down!pep who can spot a bug on a post at 100 yards. To research!rtm for some fine ideas taken from his version. To teklabs!stevenm for compiling the buglist. To vortex!lauren for always being able to see the other side of an issue. To ulysses!smb for doing most of the work involving 4.1c/4.2bsd. A special note of appreciation to bellcore!mel for dreaming up such a system that would keep so many people busy for so long improving it. And an extra acknowledgement to sfwoo!dan for hacking UUCP with such enthusiasm after these many years and for being able to tell the rest of us what the variable names meant.

References

1. D. A. Nowitz and M. E. Lesk, "A Dial-Up Network of UNIX Systems". UNIX Programmer's Manual, Seventh Edition. Volume 2B. January, 1979.
2. D. A. Nowitz, "Uucp Implementation Description". UNIX Programmer's Manual, Seventh Edition. Volume 2B. January, 1979.
3. D. A. Nowitz and M. E. Lesk, "Implementation of a UNIX network". Computer Communications, Vol 5, No 1. February, 1982.
4. D. A. Nowitz, P. Honeyman and B. E. Redman, "Experimental Implementation of UUCP: Security Aspects". To appear in Proceedings of the Winter, 1984 UniForum Conference.

Appendix I - Sample Permissions File

```
# This entry for public login
# Use default permissions
LOGNAME=nuucp

# This for some friendly outside sites when they call us
# They each have a separate login.
# When they call, we will send queued files
LOGNAME=harpo:gummo:allegra:mhtsa:mhuxt \
  SENDFILES=yes \
  WRITE=/usr/spool/uucppublic:/usr/RNEWS

# This entry for when we call these people.
# They also can execute a couple of additional commands.
# The commands are safe, so VALIDATE is not necessary on the LOGNAME entry
MACHINE=mh3bs:harpo:gummo:allegra:mhtsa:mhuxt:pwbqq \
  WRITE=/usr/spool/uucppublic:/usr/RNEWS \
  COMMANDS=rnews:rmail:xp:lp

# This entry for machines in our room (when they call us)
# The sites that login with these login-ids have extra command
# privileges, so VALIDATE name vs login-id
# (See next entry--the MACHINE values are related to these VALIDATE values)
LOGNAME=uucp:uucpl \
  VALIDATE=raven:owl:hawk:dove \
  REQUEST=yes SENDFILES=yes \
  READ=/ WRITE=/

# This entry for machines in our room -- when we call them
# It also specifies the commands they can execute locally.
# (The uucp command in COMMANDS option permits forwarding.)
MACHINE=owl:raven:hawk:dove \
  REQUEST=yes \
  COMMANDS=rnews:rmail:xp:lp:uucp \
  READ=/  WRITE=/

# This entry to call back on our faster link
LOGNAME=uucpm MACHINE=mhwpf \
  COMMANDS=rnews:rmail:xp:lp \
  CALLBACK=yes
```

Appendix II - Output From *uucheck -v*

```
*** uucheck:  Check Required Files and Directories
*** uucheck:  Directories Check Complete

*** uucheck:  Check /usr/lib/uucp/Permissions file
** LOGNAME PHASE (when they call us)

When a system logs in as: (nuucp)
        We DO NOT allow them to request files.
        We WILL NOT send files queued for them on this call.
        They can send files to
           /usr/spool/uucppublic (DEFAULT)
        Myname for the conversation will be yquem.
        PUBDIR for the conversation will be /usr/spool/uucppublic.

When a system logs in as: (harpo) (gummo) (allegra) (mhtsa) (mhuxt)
        We DO NOT allow them to request files.
        We WILL send files queued for them on this call.
        They can send files to
           /usr/spool/uucppublic
           /usr/RNEWS
        Myname for the conversation will be yquem.
        PUBDIR for the conversation will be /usr/spool/uucppublic.

When a system logs in as: (uucp) (uucpl)
        We DO allow them to request files.
        We WILL send files queued for them on this call.
        They can send files to
           /
        They can request files from
           /
        Myname for the conversation will be yquem.
        PUBDIR for the conversation will be /usr/spool/uucppublic.

When a system logs in as: (uucpm)
        We will call them back.
```

** MACHINE PHASE (when we call or execute their uux requests)

When we call system(s): (mh3bs) (harpo) (gummo) (allegra) (mhtsa) (mhuxt) (pwbqq)
    We DO NOT allow them to request files.
    They can send files to
        /usr/spool/uucppublic
        /usr/RNEWS
    Myname for the conversation will be yquem.
    PUBDIR for the conversation will be /usr/spool/uucppublic.

Machine(s): (mh3bs) (harpo) (gummo) (allegra) (mhtsa) (mhuxt) (pwbqq)
CAN execute the following commands:
command (rnews), fullname (rnews)
command (rmail), fullname (rmail)
command (xp), fullname (xp)
command (lp), fullname (lp)

When we call system(s): (owl) (raven) (hawk) (dove)
    We DO allow them to request files.
    They can send files to
        /
    They can request files from
        /
    Myname for the conversation will be yquem.
    PUBDIR for the conversation will be /usr/spool/uucppublic.

Machine(s): (owl) (raven) (hawk) (dove)
CAN execute the following commands:
command (rnews), fullname (rnews)
command (rmail), fullname (rmail)
command (xp), fullname (xp)
command (lp), fullname (lp)
command (uucp), fullname (uucp)

When we call system(s): (mhwpf)
    We DO NOT allow them to request files.
    They can send files to
        /usr/spool/uucppublic (DEFAULT)
    Myname for the conversation will be yquem.
    PUBDIR for the conversation will be /usr/spool/uucppublic.

Machine(s): (mhwpf)
CAN execute the following commands:
command (rnews), fullname (rnews)
command (rmail), fullname (rmail)
command (xp), fullname (xp)
command (lp), fullname (lp)

*** uucheck: /usr/lib/uucp/Permissions Check Complete

Appendix III - Sample Devices File

```
CALLER    LINE    USEFUL    CLASS    DIALER       CHAT SCRIPT
# the ACU's
#
# 212/801 dialers
ACU       cul0    cua0      1200     212          unused
ACU       cul1    cua1      1200     212          unused
# VenTel dialer
ACU       vn0     unused    1200     ventel       unused (for now)
ACU       vn0     unused    300      ventel       unused (for now)
# Vadic dialer
ACU       vd0     unused    1200     vadic        unused (for now)
# special entry for Vadic only systems
ACU       vd0     unused    V1200    vadic        unused (for now)
#
# the Micom also has some VenTels
ACU       Micom   secret    1200     micomventel  unused (for now)
ACU       Micom   secret    300      micomventel  unused (for now)
#
#
# the switches
#
# Micom pbx
# 4800 baud is funny ...
Micom     mc0     unused    4800     unused       ' \s\c NAME? %s GO \c
Micom     mc0     unused    Any      unused       '' NAME? %s GO \c
Micom     mc1     unused    4800     unused       ' \s\c NAME? %s GO \c
Micom     mc1     unused    Any      unused       '' NAME? %s GO \c
# Develcon pbx
Develcon  dv0     unused    Any      unused       '' Request: %s \007 \c
Develcon  dv1     unused    Any      unused       '' Request: %s \007 \c
# DATAKIT PS
Datakit   dk0     unused    Any      unused       ' \d%s
Datakit   dk1     unused    Any      unused       ' \d%s
# gandalf
Gandalf   gd0     unused    Any      unused       '' class %s start \c
Gandalf   gd1     unused    Any      unused       '' class %s start \c
```

Appendix IV - Sample Systems File

SITE	WHEN	CALLER	CLASS	CALLCODE	LOGIN
fonzie	Any	ACU	D1200	MHd1234	...
fonzie	Any0631-0444	ACU	C1200	MH5678	...
fonzie	Any0631-0444;5	Micom	Any	fonz	...
fonzie	Wk1800-0600,Sa	Datakit,dg	unused	fonzie	...
fonzie	MoWeFr1300-1445	Ethernet,eg	unused	09	...
fonzie	Any	Direct	9600	tty42	...

Appendix V - Example of a Uuxqt Error Report

remote execution [uucp job allegraA60f1 (6/6-2:50:32)]
rnews
exited with status 1

= = = = = stderr was = = = = =
rnews: Cannot open /usr/spool/news/.sys (r) (From: harpo!decvax!pur-ee!iuvax!dcm).
perror: No such file or directory

= = = = = stdin was = = = = =
From: harpo!decvax!pur-ee!iuvax!dcm
Newsgroups: net.unix-wizards
Title: uucp trap 9 ever fixed?
Article-I.D.: iuvax.119
Posted: Fri Jul 2 11:48:26 1982
Received: Sat Jul 3 00:54:31 1982

We (a VAX 4.1) are getting alot of trap 9s, probably from
the bug mentioned in rv(4); has anyone ever fixed this?
pur-ee!iuvax!dcm

EURIX - a UNIX Based System Using European Natural Languages

ir P. Tintel

Bell Telephone Manufacturing Company
Automation Division (BA40) 1064 F/J2
Francis Wellesplein 1
B-2018 Antwerp
Belgium

1. SITUATION OF THE PROJECT

The UNIX operating system has gained an enormous popularity. It is used in many places for software development, but it is also beginning to be used in office environments. However, the average office worker is no computer specialist and has other demands concerning the system than software developers.

The UNIX user interface as it is today has certain drawbacks for office applications and more specific for office applications in Europe:

- UNIX has a very terse command set. It is not very helpful when you want to find out how to use certain commands. The manuals are not very readable for someone not familiar with UNIX.
- All communication with the user is performed in English (or American). This is fine for software developers who prefer this, but is not suitable for the office workers in Europe, who are not familiar with the English language.
- UNIX is not capable of working with the different European characters. The ASCII character coding, as adopted in UNIX, does not allow coding of, for instance, the French accented e's (é, ê, ë).

2. SOLUTIONS TO THE PROBLEM

The EURIX project focuses on the second and third problem: EURIX will communicate with the user in the user's natural language and will be able to handle the special European characters in a uniform way.

2.1. Communication with the user

When the system must communicate with the user in the user's natural language, commands must generate messages and prompts in that language. To achieve this, a few possible approaches can be taken:

- Provide several versions of each program, each version using one particular natural language. A disadvantage of this approach is that there will be a lot of duplicate programs. Also, when there is no source code available, programs cannot be adapted to a new language.
- Parametrize the messages and prompts of the program. Every set of parameters will represent one language. With one set of parameters, made up for a particular natural language, a program behaves as if it were written for that language. A new language can be introduced by providing a new set of strings, even without the source code of the program being available. This approach looks preferable.

2.2. Character coding

In this section, a clear distinction is made between a character set, and a code set. A character set is the set of graphically representable characters (example character 'a', character '0', character ë'). A code set is the set of codes that a device will understand (example hex25, hex6A, ...). A device defines a mapping from a code set to a character set.

For the introduction of special European characters, there are a number of possible solutions:

- In the ASCII character set a number of rarely used characters can be redefined as special European characters. This is the approach taken by most of the terminal hardware manufacturers, but there are some basic problems with this approach:
  - There are not enough codes to represent all the special European characters. So the need arises for several characters sets, each supporting one language. All these character sets share the same code set, so there is no way to distinguish between various character sets when looking at the code of a text.
  - Texts prepared with one character set cannot be displayed (or printed) correctly on devices that do not have the same character set. Since UNIX is an open system, one can expect lots of different devices connected to it and so, the interchangeability of texts is limited.
  - The fact that characters are rarely used does not mean that they are not used at all. They may still be necessary under certain conditions (e.g. French comments in Cprograms).
- The special European characters can be represented by means of escape sequences or with a shift to a character set that contains them. The explicit shifts between character sets complicate application programs considerably. Also, there is little standardization concerning these character sets.
- The standardization institutes (CCITT, ISO) propose a new code set, using an eight bit code (ASCII uses only seven bits). CCITT has adopted the TELETEX standard (recommendation S.61) which is used for an elaboration of the telex services. This recommendation is mainly concerned with text, as opposed to some other standards, such as NAPLPS (*), that also include graphical elements North American Presentation Level Protocol Syntax. UNIX is not ready yet for a general introduction of graphical elements.

The TELETEX character set is a superset of the ASCII character set, and the TELETEX code set is a superset of the ASCII code set. In addition the TELETEX character set provides the special European characters. The code set contains some non spacing codes (underline and diacritical marks). Non spacing codes, together with a spacing code form one character by 'overstriking'.

In EURIX, the TELETEX character and code set have been adopted. These sets allow for the direct representation of all the European characters.

3. IMPLEMENTATION

In this section the three aspects of the implementation of a system with parametrized strings and TELETEX character set will be described.

3.1. String data base

The parametrization of the natural language used by a program is implemented via a string data base. This data base is structured hierarchically. For every program there are files that contain the strings of the program in a specific natural language. For every language there is such a file. For efficiency reasons these files exist in two forms: a textual form (source) that is intended for human use and a binary (compiled) form that is used by the system. These files define an equivalence between a unique number and a string. A conversion program between the two representations is provided.

In a program, the occurrences of strings are replaced by a call to a routine 'string()', with the string number as argument. The routine will return a pointer to the string corresponding to that number. Strings used as initializer of globally defined string variables cannot be replaced by a mere call of the routine 'string()'. These strings initializers are removed and, at run time, the variables are initialized explicitly. This initialization is done by the routine 'sdbinit()', which is called before the 'main()' procedure is entered.

A special function 'error()' is available for the output of error messages, giving a classification of the error, and possibly the termination of the program with a certain exit code. The classification and, when needed, the exit code are taken from the string data base file and so can be modified

without recompiling the program.

When a command starts execution, it will first determine what string data base file to use. Next, it will load all its strings and error messages, and initialize the globally defined string variables. Finally it will start the main program. The string data base file to be used is taken from a search path, which is constructed in a way similar to the command search path of the shell. This search path is defined by the 'STRINGDB' environment variable.

For the introduction of the string data base, almost every program needs modifications. It is therefore useful to perform those modifications mechanically, e.g. by a preprocessor. Based on this preprocessor, a special compiler generates an adapted object module, together with a string data base file. In this way, the string data base modifications are kept independent from the sources. The sources themselves are not modified, which is important for the follow up of future releases.

The introduction of a new natural language comes down to the translation of the strings and error messages of the programs. This translations need to be done manually, but is independent of the program itself.

3.2. TELETEX character set

There are two major categories of modification necessary for the introduction of the TELETEX character and code set: modifications to the kernel and modifications to applications.

In the kernel, a way to display the TELETEX characters on a normal ASCII terminal is implemented. The codes with the eighth bit zero (ASCII codes), are passed as they are, and the codes with the eighth bit one are escaped. The escape sequence is similar to the one used in UNIX to distinguish between upper and lower case characters for terminals that cannot make this distinction themselves. When, on input, a character is preceded by a back quote (`), the system will add the eighth bit. On output, a character with the eighth bit set will be preceded by a back quote. The special meaning of a character can be escaped with a back slash (\ ).

For the introduction of the TELETEX character set in application programs, the following types of modifications can be distinguished:

- The eight bit TELETEX code set is introduced. When a seven bit code set is used, a character code stored in a byte leaves one bit free. Some programs (editor, shell,...) originally used that free bit to store some some additional information concerning the character. When all eight bits are used for the character code, this additional information need to be stored in an other way:
  - An escape sequence can be inserted before the character. When there are codes in the code set that are not used to represent valid characters (as is the case in the TELETEX code set), the escape sequence can be one of those. When the code starting the escape sequence is itself a valid character, a more complicated escape sequence is necessary. As this additional information is binary of nature, the mere existence of the escape sequence indicates the additional information.
  - Space for the extra information can always be foreseen. A character will then be stored in two bytes, one with the extra information and one with the character itself. This scheme is desirable when direct access to some character is necessary. A disadvantage of this scheme is a considerably larger memory use.

  Note that these changes are completely local to single programs. In files the TELETEX codes are used.
- Programs that treat the contents of files (editors, compilers,...) need to know about the new character set. These programs have a larger set of valid character codes and need a classification of the new characters.
- Formatters, such as nroff require addition semantics for the handling of languages other than English. The entire hyphenation mechanism needs modification in order to perform correct hyphenation for the various European languages.

- Formatting programs need to know about the existence of non spacing character codes.

3.3. Terminal hardware

At the moment there are no TELETEX terminals available with RS232 connection. An existing ASCII terminal (Perkin Elmer 1251I - INtexfII) has been adapted to accommodate a major part of the TELETEX character set. This terminal generates and accepts the 7 bit escape codes as used by the host for emulation of TELETEX characters on an ASCII terminal.

The main problem with the terminal is the way in which character images are stored. There is only room for 256 images, while the TELETEX character set contains over 300 characters. Also, the generation, from the keyboard, of TELETEX characters is not very elegant. To generate a code with the eight bit set, two key strokes are necessary. The worst case, an underlined character having a diacritical mark, needs five key strokes.

4. EXPERIENCES

In the implementation of the TELETEX character set for VAX and PDP, special care had to be taken of sign extension. These sign extensions arises whenever a character is used in arithmetics or in comparisons with numeric constants.

The string data base does not seem to introduce a noticeable overhead for normal commands. Only the initial phase causes some delay for commands that need a lot of strings.

The automatic introduction of the string data base saved a lot of programming effort. The translation to other languages is not always trivial, as proper terminology for UNIX concepts is not always available in that particular language.

5. CONCLUSION

With the EURIX system, the language aspect of the user interface of UNIX is adapted for the use by Europeans. Other aspects such as the name of commands and options of commands still need attention. Also a more verbose user interface is desirable.

New terminals that understand the TELETEX character set are necessary.

Large Systems UNIX Opportunity for Innovation

Carol Realini
Andy Tucci

Amdahl Corporation
Havenweg 24
4131 NM Vianen
Netherlands

ABSTRACT

This paper will discuss the issues involved in making UNIX a viable mainframe operating system. These include operations management, reliability, communications, performance, security, database systems, applications software, compilers, and coexistence with other systems. The issues are many times related to the expectations of the System/370 mainframe user who is currently running VM/370 (VM) and MVS. Other issues surface on a mainframe UNIX because the mainframe environment is different from the mini-computer or micro-computer environment. In bringing UNIX to the System/370 mainframe world, there is a merging of two standards: the UNIX standard of openness, ease of use, and ease of development, and the IBM mainframe operating system standard of high reliability, security, and performance. Combining the standards is a challenge.

1. INTRODUCTION

Although UNIX was originally a program development system for small groups of users, we are beginning to see large development projects with many users on a UNIX system. In addition, UNIX is also being used as a production operating system. These changing uses of UNIX have created a demand for a large capacity UNIX system: one which supports many users, runs many different applications, and shares many critical resources. This environment requires UNIX to have features, functions, and characteristics that are not necessarily important in other more traditional uses of UNIX.

2. OPERATIONS MANAGEMENT

Operating a mainframe is much more complex than operating a small system. The larger number of users, the many types of peripherals, and the large amount of shared resources all contribute to making the mainframe operation more difficult to manage than micro-computer or mini-computer operation. The mainframe installation expects the operating system to provide many tools and capabilities to help the operations staff meet the demands of this complex environment. Specific areas which require attention are accounting, tape management, the operator interface, system maintenance, and resource management.

2.1. Accounting

Accounting may not be too important on a small or medium size systemwhere the budget is modest or where it may be possible to assign the cost of the whole system to one project or department. Accounting becomes more important to the mainframe DP manager because the hardware and software represent very large expenditures and because the resources are likely shared by many users. The DP manager needs to be able to allocate the cost of various resources to the projects, workgroups, departments, or organizations using the resources. Therefore a mainframe operating system must be able to collect detailed data about the use of resources, the data must be accurate and complete, and report generating capability needs to be provided.

Accounting records need to be kept for all resource utilization including CPU usage, disk space, tape and mountable disks, printed output, terminal usage, user connect time, and main memory usage. These records can be made at the time a process uses a resource-- one record for each disk access, or at the completion of a user session or process--or information can be collected during the user's session or process, with records written at session or process termination time. Both schemes can be used; we recommend using a combination. Whenever the latter is used, special care needs to be taken to record accounting information in the case of abnormal termination.

Each installation has a need for different accounting reports. They need the reports to be configurable to their situation, easy to run, and easy to understand. Data and reports should allow for the summaries of resource usage by work group, department, organization, or project.

Some installations want to combine UNIX accounting records with VM and MVS accounting records and then to use a single data analysis and report generation process. The most common System/370-compatible mainframe accounting records are MVS 'SMF' records, and most System/370-compatible hardware shops have already invested the effort to develop programs and procedures to analyze and report the information. A UNIX accounting to 'SMF' translator would give many installations a single data analysis and reporting system while taking advantage of all the effort they have already put into their own systems. We are not suggesting that this approach replace providing the function in UNIX, but that it be another solution available to the DP manager with the understanding that some may choose to use it.

2.2. Tape Management

In a small system environment there may be very few or no tapes and tape handling is simple and straight forward. A typical mainframe installation is more likely to have thousands of tapes, and many procedures for handling tapes. Many problems can occur with large numbers of tapes and many different handling procedures which can be addressed with tape management capabilities in the operating system. The capabilities include label checking, expiration date processing, and tape library facilities.

Label checking prevents accidental or intentional misuse of system and user tapes. Unique labels on tapes identify the tape and that identification can be associated with the user and use of the tape. When the tape is used, checking that the user is allowed to use this tape for this purpose helps prevent the misuse and loss of data.

Expiration dates for tapes allow an installation to automate some or all of the process of reclaiming tapes after their usefulness has expired. Many installations do not automatically reclaim tapes once they have expired but use the tape expiration reports to trigger a request to the user for permission to reclaim the tape. In some cases no response after a set time is interpreted as permission.

For large tape libraries there is a lot of maintenance of the library that could be enhanced by operating system tools. Expiration, reclaiming, and consolidation of tapes can usually be procedduralized for an installation and UNIX could provide tools and facilities to automate the procedures.

Both VM and MVS provide only very basic tape management facilities. Most installations purchase add-on products to augment the system facilities. The above suggested capabilities could easily be built into UNIX and we believe they would receive a very positive response from the DP manager, who is probably wondering why he has to purchase that additional package and why the functionality is not in the operating system he has today.

2.3. Operator Interface

UNIX on a minicomputer is, in most cases, operatorless. A mainframe UNIX system can require one or more operators. The operator interface requires new functions and a new kind of user friendliness not necessarily needed in the past.

The functions include commands and procedures to accomplish the following tasks: tape and disk movement, system backup and recovery, incremental backup and recovery, system initialization and termination, dynamic device configuration, and hardware problem detection.

The typical operator is not a systems programmer or a programmer. The operator does not have a programmer's knowledge of the shell and the commands, nor does he have a conceptual understanding of UNIX. Without these skills and this knowledge the operator can get into trouble as an ordinary user and a lot of trouble as a 'superuser'.

A very controlled environment is in order. The 'superuser' privileges should be under very controlled use; the privileges should be available only through programs that perform the tasks. See discussion of setting userid root programs in the Userid Administration topic of the Security section. In addition, not all capabilities of the shell and not all commands should be available to the operator. Piping and redirection are advanced concepts and can get even an experienced user into trouble. For example, redirecting in the wrong direction can destroy files. A limited shell and command set prevents many possible errors from occurring.

In designing the operator interface it is critical to understand the operator's job and build commands and procedures that map to his understanding of his job. Additionally it is a good idea to look at the VM and MVS operator interface and design the UNIX mainframe operator interface to be comfortable for operators who must split their time between UNIX, VM, and MVS and/or are used to the VM and MVS operator commands.

2.4. System Programming and User Support Tools

VM is an excellent tool for the systems programmer, and can be one of the bonuses that come with an implementation of UNIX for the System/370 architecture. To all the systems programmers who work during the middle of the night to test their new kernel, it is a relief to have a virtual machine available for testing at all times. It gives them all sorts of flexibility they formerly didn't have including the ability to have various test systems at one time.

In addition to system maintenance responsibilities, systems programmers, along with the user support people, have a responsibility to help users with their problems. The tracking of user problems on a UNIX system with hundreds of users can be difficult. It is extremely useful to have a simple, easy to use problem reporting and tracking system that helps keep track of the user's problems, requests for enhancements, and questions.

2.5. Resource Management

Resource managementwithin a computer system is a complex subject; many PhD theses have been written on a single topic within the subject. It probably at least deserves a separate paper, but let us touch on the subject here in order to explain our general direction and areas of focus.

A good philosophy is "Provide the basic function needed but keep it simple". The problems of managing large amounts of resource, balancing the loads, preventing heavy users from impacting others, and using resources efficiently are non-trivial to solve. Yet we refuse to believe it takes hundreds of thousands of lines of code and 30% of a mainframe CPU to come up with acceptable solutions, as is the case with MVS.

We do feel additional functions need to be added to UNIX to deal effectively with resource management on a mainframe. The new functions required are the facilities to manage scheduled deferred work queues, the ability to limit the resource utilization of a user or group of users, the ability to partition portions of the resources and allocate them to user groups so they can manage them, and improved scheduling techniques within the kernel.

Adding the functions and keeping the system simple is possible, but tradeoffs have to be made. It is important to remember that machine resources are getting cheaper everyday and people resources are getting more expensive and harder to find. Every effort should be made to keep the user interface simple, to make the resource management facilities easy to use, and to meet the needs of the operations management staff. This will save programmer, system programmer, and operation staff time, which in most cases is more important than saving a few instruction cycles or a few cylinders of disk space.

2.6. Operations Management Summary

The above sections describe many opportunity areas for UNIX. Developing easy to use, simple tools for the operations staff complements UNIX's already outstanding programmer tool set and programmer productivity oriented environment. Although some new facilities would take time to develop, others could be built quickly and, if done well, would enhance UNIX's ease of operation on a mainframe.

3. RELIABILITY

The mainframe DP manager expects the hardware and operating systems software to be extremely reliable. Not only does he have a large user community which is impacted by system failures, but he is also used to having MVS recover from many failures and isolate other failures to particular users.

Many mainframe installations aim to provide 100% computer resource availability; some installations are today achieving 98-99% availability levels. UNIX must attempt recovery from hardware and software errors, provide adequate error logging and reporting, and provide diagnostic tools that can be used stand-alone as well as while the system is active.

UNIX needs to recover from hardware errors if possible. I/O errors should be isolated to the user or system process affected; the system need not fail unless the superblock of a disk is damaged. I/O robustness is a must for the kernel; the kernel needs to retry I/O or do whatever else is required to continue operation. Recovery can include taking the device offline, signaling the appropriate processes, or whatever else is needed to increase the availability of the computer resource. If a problem can be isolated to one of the disks in a multiple-disk file system, it is necessary to be able to take only that file system off line.

As a short example, when a page in memory takes a parity error, the hardware reflects that error to the operating system. The kernel must not 'panic' if the page is in a user's program space. The user process should be killed, the page in memory needs to be invalidated, and the kernel should keep on running.

Error logging and reporting is crucial to system reliability. Reporting recoverable errors facilitates preventive maintenance activities, which are important in avoiding unrecoverable errors. In a System/370-compatible mainframe installation the reports generated should be compatible with VM and MVS error reports. Vendor hardware maintenance engineers are familiar with those reports and know where to find the information they need. If the error report format is different, they may have problems finding the right data.

Running diagnostics while other system activities continue is a requirement for all large data processing shops. With hundreds of users it is not practical to take the entire system down if problems can be isolated to a particular file system. Additional UNIX software is required for isolation and resolution of these kinds of problems.

In summary, the kernel needs robust error handling, a proper error reporting mechanism, and good diagnostic capabilities. Error should affect as few users as possible. Much effort should be made to minimize the impact of non-recoverable errors and to provide support tools used to do preventive maintenance which helps avoid non-recoverable errors.

4. COMMUNICATIONS

In bringing UNIX to System/370 mainframes, extensions are required for communications. These include communications with other UNIX systems, communications with VM and MVS, sharing large networks of devices, connecting the mainframe with microcomputers and minicomputers, and sharing data from large databases on the mainframe with the applications running on workstations.

Communications with other UNIX systems is provided by standard in UNIX. The System/370 implementation need only take these functions and provide for System/370 communication links.

To communications with VM and MVS, UNIX needs to provide the ability to transfer files, jobs, and output to and from MVS job queues and VM virtual machines.

VM and MVS support the IBM System Network Architecture, SNA, which allows sharing of large terminal networks. SNA support could be built into UNIX but is a very large effort. For now, SNA devices are only available from MVS or UNIX running under VM with a service virtual machine supporting SNA.

Additional security issues arise when the mainframe is connected to networks of other systems. These issues need management attention and visibility. There is a higher probability of sensitive information on the mainframe, just because of the greater number of users using a wide range of applications, so the information is easier to pass around if proper understanding and controls are not in place.

Communications to other systems and networks is important for a mainframe operating system. Some capabilities already exist, especially for UNIX-to-UNIX communications. Other areas need a lot of research and development to catch up with VM and MVS.

5. PERFORMANCE

Performance on a mainframe is an area where innovation can be very productive. The high-powered processors, large main memories, and high-speed peripherals of mainframes translate into a highly performing UNIX without any major system changes to efficiently utilize the resources. Additional performance gains can be achieved with system enhancements.

The size and power of today's mainframes is impressive. The processor range from 5 mips to 20 mips. Main memory can be up to 64 megabytes and very fast paging devices can be used when main memory runs out. This translates into fast response times, large numbers of users on the same processor, and very short elapse times for programs. For example, a rebuilding of the kernel can take all night on a microprocessor, a couple of hours on a minicomputer, and can be done in approximately 10 minutes on a Amdahl 5860 processor.

Additional performance gains can be achieved by enhancing the UNIX process scheduler, optimizing the code generated by the C compiler, and improving paging algorithms. The UNIX process scheduler is acceptable for minicomputers and microcomputers, but not as effective for hundreds of users using large amounts of resources. Research and development work in this area would definitely reap benefits for UNIX vendors. Any improvements to the C compiler code generator improves system performance and C application performance. The paging algorithms can be modified according to the types of work being done on the mainframe. We suggest that tuneable parameters be added to allow for the tuning of the kernel, depending on paging load.

6. SECURITY

Security is an extremely important issue in a large system environment. The security administrator needs to be able to restrict access to resources and data to monitor and track the use of resources, and to be confident that the system prevents security violations. UNIX today meets only some of these requirements. The challenge ahead is to provide additional functions, monitoring, and controls without sacrificing the simplicity of UNIX and the ease with which information can be shared in a UNIX system.

A mainframe operating system needs to allow the installation to be as serious or lax about security as required. Capabilities should be flexible and configurable. If new security controls are needed, our experiences tell us they can be difficult to introduce. The users may perceive the new controls as inconvenient, uncalled for, or unimportant. They may feel a right is being taken away. The change needs to be smooth and as painless as possible; the system should be easy to use and implementable in stages (a few small changes may be more acceptable than one big change).

6.1. Userid Administration

Along with large numbers of users comes the problem of managing large numbers of userids. On a minicomputer UNIX with 40 users, the administrator probably knows each user and can easily manage the userid and personnel changes. On a large mainframe with as many as 1000 userids, the userid administration becomes more complicated and the systems administrator needs tools to help understand and monitor userids.

Programs that generate reports about the /etc/password file can be very helpful. The reports can flag inactive userids, activity on userids for people who have left the company, and all userids that require changes. Categories of userids could be defined and certain categories can get special treatment. For example, 'contractor' could be a category and those userids could expire after a set interval, requiring authorization to reactivate after the contractor's relationship with the company is reviewed.

Userid administration is time consuming for a large UNIX system. The tasks are usually delegated to a non-technical individual; the UNIX systems programmer is not interested in spending many hours a day doing userid administration. In very large installations, there may be multiple userid administrators, each having a set of userids to maintain.

Although the responsibility for userid administration is delegated to a non-systems programmer, it is not desirable to allow this individual to be a 'superuser'. We recommend thoroughly tested programs that use the 'set userid root' facility. The administrator has regular user privileges and the 'superuser' privileges needed to maintain userids while these special applications are running. The program source is read-protected to prevent a user from understanding how it works and discovering a security hole. It is write-protected to prevent users from modifying or replacing it with their source. In addition, interrupts are trapped to prevent exit from the program before the 'superuser' state is switched off. For multiple userid administrators we recommend facilities to define sets of userids and restrict the administrators to their assigned groups. In both cases, we recommend a user-friendly interface that is comfortable for a non-technical person to use.

6.2. Data Security

Data security can be extremely important to large installations. Engineering companies are becoming extremely concerned about protecting their hardware logic diagrams. Software developers are interested in protecting their trade-secret source code. It is no longer just the banks and payroll data that need to be protected.

In many UNIX systems today, the default is to share all files with other users. The reverse may be desirable at some installations so that sharing requires an action on the part of the user. Files would then be explicitly shared with individuals or groups. To make this scheme work, sharing with specific individuals or groups must be kept simple, otherwise people, out of frustration, might open their files to everyone to avoid the bother.

6.3. Audit Trails

Logging of various events provides audit trails which can be used to identify security violation. The type of events logged and the frequency of the logging depends on the requirements of the installation.

Detailed audit trails of user sessions and object access can be important in resolving incidents. If something is stolen or deleted, these records are evidence of the crime. The cost, of course, is computer time and disk space to collect and save all the information. In most installations very few objects would require this journaling. Yet for important files, the cost may be justified.

One simple and useful log is to save the time and physical terminal address each time the userid was used. Inactive userids can then easily be identified, and the use of userids during vacations or after termination can be found. Logging the installation or modifications of commands allows auditing of system changes. The 'superuser' sessions could be journaled and monitored daily to detect violation quickly. An installation might choose to monitor late-night sessions for

suspicious activities. Password violation logging can detect attempts to guess passwords through trial and error. A tolerance level can be set, after which the userid or physical terminal is deactivated, requiring reactivation by the system administrator. Both are deterrents that installations might choose to put in place.

6.4. Security Summary

In conclusion, we recommend adding functions to UNIX to provide the capabilities described above. The emphasis should be on flexibility, user friendliness, ease of use, and trustworthiness. The system should be able to be configured to the requirements of the installation and be changeable as the requirements change.

7. SYSTEM AND APPLICATIONS SOFTWARE

For operating systems vendors it is desirable to have as much application software as possible available for the operating system. The more software, the more choice the user has and the higher the probability that among the choices is software that meets the user's expectations. The mainframe user's expectations are high since VM and MVS, because they have been around for years, offer a great number of mature systems and applications software packages.

System software and applications software is becoming available for UNIX at a rapid rate and the rate is being accelerated due to two factors. The first factor is the marketability of UNIX software; software vendors are becoming extremely conscious of the potential in the UNIX marketplace. The second factor is the move by software vendors toward operating system 'independence'. Vendors are acutely aware of the high price they have paid in the past of having a product which is difficult to port to different operating systems. More and more, application software is implemented without tightly connecting it to the operating system. Lots of vendors who previously optimized their software for an operating system are now expending much effort to isolate the operating-system-dependent code and port the product to different operating systems. This 'independence' movement means that non-UNIX software can be more quickly and easily ported to UNIX.

Compilers and database management systems are product areas where software for UNIX has not caught up to VM and MVS. Certain compilers are not yet available on UNIX, while those that are available may not be as reliable, may not perform as well, or may not provide as many functions as the VM and MVS compilers. Several good Relational database products are available for UNIX, and we applaud their flexibility and end-user accessibility; they are well suited for information center applications. It is yet to be shown if they can displace the need for high- transaction-oriented, highly structured databases such as IMS. It is also questionable how large of a capacity can be supported effectively for a production database. Our sense is that it will take some maturing of these database manager systems before they can manage a wide range of mainframe database needs.

With all the UNIX software development currently underway we will be seeing many more software packages available in the future.

8. COEXISTENCE WITH MVS, VM, AND UNIX

No matter what your personal preference in operating systems, VM and MVS are here to stay for System/370 compatible mainframes, and any operating system for System/370-compatible hardware needs to be able to coexist with them. A mainframe UNIX must acknowledging this need and build tools and facilities to facilitate this coexistence.

In many mainframe installations work will be divided between UNIX, MVS, and VM. In some installations the development on UNIX has roots or old branches on VM or MVS, which is not cost effective, desirable, or even possible to move. At other installations UNIX may have been chosen to implement only part of a large system whose other parts are on VM or MVS. It is only a very few existing MVS or VM shops that will eventually phase out VM and MVS.

Two elements to coexistence are communications and compatibility. Providing the facilities that we discussed in the communications section, will be important for a mainframe UNIX.

Compatibility between systems can beprovided at different levels: functional, language source, or object. The level of compatibility depends on the requirements for the situation.

Functional compatibility means that a function in MVS or VM has an equivalent function in UNIX, but the function in UNIX does not necessarily have the same name or syntax. For example, the command to copy a file is not the same under UNIX as it is under VM or MVS but the capability is provided on all three systems.

Language source compatibility means language source definitions are the same across systems. Programs would need to be recompiled but would produce the same results across systems. Language source compatibility is useful if application development crosses operating system boundaries. It is attainable if the UNIX language and MVS or VM language definitions are very similar.

Object compatibility allows the same object files to be loadable and executable in different systems. Object compatibility is desirable (but expensive to implement) for programs where source is no longer available and/or reimplementation or porting would be expensive.

Tools and facilities to allow UNIX to smoothly coexist with VM and MVS are important to mainframe UNIX. Installations with VM or MVS and UNIX will need them to varying degrees depending on how much the projects span operating systems boundaries.

9. SUMMARY

Having spent most of our careers working on software for an System/370-compatible hardware manufacturer, we have seen the IBM operating system software world mature. In many ways, UNIX is where VM and MVS were 10 years ago. In other ways UNIX was designed from the start with features and capabilities that IBM is still attempting to add onto VM and MVS. We would like to see the UNIX community take advantage of what was learned from VM and MVS and what was built into UNIX from the start to move UNIX quickly into a position of being a viable large-system operating system.

Acknowledgements

A special thanks for the ideas, inspiration, and support from Cyndy Ainsworth, Carole Beckham, Ken Hoadley, Eric Corwin, Donal O'Shea, Henry Wacker, Colin White, and the UTS Products Group.

Behind Every Binary License is the UNIX Heritage

Brian E. Redman

Central Services Organization
Whippany, NJ 07981

(ber@yquem.UUCP)

Pat E. Parseghian

Princeton University
Princeton, NJ 08544

ABSTRACT

Lately there seems to be some pessimism about the future of the UNIX system. Many who have watched its development from the earliest days feel that the system appears to grow corrupt and is no longer a model of innovation in operating system design.

UNIX was originally designed by a talented fraternity with a clear and common vision for a better computing environment. Ever since, the system has been redesigned by a diversity of people with different goals that tend to be less clear. UNIX has evolved from a simple, elegant model into one that is certainly complex and often seems convoluted. It no longer constitutes a statement of smallness, but appears to be growing unrestrictedly. It is generally accepted that the original systems provided a rich environment for a community of sophisticated computer users, as was intended. More recently it seems that UNIX is expected to be a computing panacea, and the compromises that have increased its palatability (and indeed, popularity) have reduced its effectiveness for its initial application.

One important difference between systems of the past and those that we'll see in the future is a preponderance of "binary only" applications. It is disconcerting that the "total system" may no longer be distributed, or may be available only at high costs.

The term UNIX has come to represent more than an operating system or computing environment; it represents philosophies about computing. Although the UNIX community may question the costs and motivations underlying these changes, we feel it is critical to recognize the important benefits that have been realized: UNIX and its philosophy have been spread among the computing masses and have influenced the direction of computing. The commercialization of UNIX is largely responsible for this. Other systems may have been just as revolutionary, but will never have a similar impact because they were kept private.

The following opinions are our own and are not likely to reflect those of our former employer, AT&T Bell Laboratories.

This paper is about "the cheese". Figure 1 is a reproduction of a poster that reminds us of the "proper" usage of the term UNIX. The usage dictated by these rules is a legal interpretation of a word that symbolizes many more ideas than those enumerated.

Throughout this paper we represent the sentiments and ideas of many people. There has been a great deal written and said about the UNIX system and our opinions reflect that. We only first heard about the UNIX time-sharing system as students several years after its description was published in the Communications of the Association for Computing Machinery [1]. We did not experience its infancy, but one need not live through a period in order to appreciate it.

In part, UNIX is an operating system that can be characterized by its primitive operations. These include *read, write, open, close, fork* and *exec.* Clearly a system lacking these functions is not UNIX (though some purport to be). The right collection of system calls and their proper behaviour is necessary, but is by no means sufficient. At another level, UNIX is represented by the commands that one expects to find. We tend to distinguish between commands that are building blocks, designed to be fitted together to form new commands, and those that are subsystems or special purpose objects used in and of themselves. *Grep, sort, sed* and *tr* characterize the former. The **C** compiler, a news facility and some games represent the latter. By our definition a system lacking *grep* is not UNIX; likewise one without games is certainly suspect.

UNIX is also the environment in which we find it. An environment where the entire system's source is available. Such an environment is conducive to our understanding and our learning as we look at the system itself for models of good programming. This also provides us the opportunity to build on the work of others and avoid reinvention and incompatibilities. And of course source encourages us to find the causes of problems and fix them, or at least clearly specify them. The ability to move throughout the system and learn from it contributes to our better understanding of its overall workings.

Some less technical and more sociological factors also contribute to the definition of UNIX. Personalities from various universities, some government agencies and a few industrial laboratories are themselves a part of UNIX. The comments in the system's code are testimonials to these people. For example in an assembly language assist routine, uldiv.s, we find the illuminating comment, "this is the clever part". Indeed! Such style is an important part of UNIX.

Perhaps the most important aspect of this definition is the philosophy that bred the kernel, fostered the commands, and attracted the community. The philosophy of UNIX is alluded to if not defined outright in many publications by various authors (including Kernighan and Plauger [2,3], Kernighan and Ritchie [4], Kernighan and Mashey [5] and most recently in a book by Kernighan and Pike [6]). The philosophy dictates a system that is made up of small powerful functions, a system composed of the elements of programming style.

Given this definition of UNIX, how did it come about and why did it grow beyond a cult experience to legitimacy? Recently, these questions have been among those addressed by Dennis Ritchie in his Turing lecture [7] and by Kernighan and Pike in the epilogue of *The UNIX Programming Environment.* Ritchie felt that the principal technical aspect of UNIX that led to its initial popularity was that it was a simple and coherent system that pushed a few good ideas and models to the limit. As Ritchie explained, there were other circumstances that contributed to its success: it was introduced when minicomputers were first being seen as viable alternatives to large centrally administered mainframes; it was available on attractive hardware, the PDP-11; and its development was influenced by enthusiastic and technically competent users over a relatively long period of time. Kernighan and Pike assert that "The central factor is that it was designed and built by a small number (two) of exceptionally talented people, whose sole purpose was to create an environment that would be convenient for program development." Both sources cite the fact that UNIX, because of this initial popularity, became important when its original followers entered the real world and demanded that UNIX be present.

The UNIX of the early seventies was readily available to a generation of forward-looking computer scientists. It was either virtually free (through modest licensing agreements to educational institutions) or effectively free (to other interests with significant financial resources). It required modest support in terms of hardware. It was elegantly simple and could be understood by a single person. And it provided a foundation for a large share of state of the art research because of the nature of the people who tended to seek it out.

It would be pleasant if that were the end of our story. And yet that is essentially the end of the UNIX story, because the UNIX that we have described has come and gone. As that UNIX fades away into a rosy memory a new concept of UNIX takes its place in the present. We feel they're distinct. Now we'll refer to the UNIX of the seventies as the academic UNIX, and the current commercial UNIX will be the cheese.

Figure 1. UNIX Cheese

If we had to indict the moment when the academic value of UNIX peaked (and we're led to believe that we must) then we would say it was the time just before the VAX was introduced. †VAX is a Trademark of Digital Equipment Corporation. Indeed, the VAX-11/780 was a reverse Pandora's box in that it attracted evils. Thirty-two bit addressing was the harbinger of doom. Virtual memory was the crushing blow. Perhaps the constraint of the PDP-11 architecture provided the conscience that guarded UNIX.

We've had a difficult time trying to pinpoint the problems in various versions of UNIX that make us feel that something's gone wrong. We can think of various examples; the addition of uncritical system calls such as *ulimit,* the proliferation of trivial commands such as *logdir,* the overwhelming infestation of subsystems such as *lp* or *SCCS* which render a once concise manual diffuse. But these are just symptoms of an overall problem. Perhaps the best way to describe the problem is to contrast the present UNIX with its predecessor. We'll admit that people can argue convincingly that Sixth Edition UNIX wasn't perfect. But we don't support them.

Clearly the central factor underlying the brouhaha over UNIX today is money. That's a sharp contrast when we consider that had the development of academic UNIX been blessed with funds, UNIX would probably have been implemented on a PDP-10 and experienced quite a different future (had it been created at all).

We recall at the UNIX users group meeting in January, 1979 that a speaker from Western Electric talked about licensing. He answered questions, quoted policies and refused to predict the future. Even at that late date, we suppose there were relatively few commercial licensees because in June of 1980, Al Arms reported that there were twenty-four. Each time we saw the Patent Licensing manager from Western Electric in front of a UNIX gathering he reported the facts and figures. But each time, they took on more significance. Initially, UNIX licensing was a minor chore, but something the Bell System was obligated to do. Commercial licensees paid some substantial fees, yet they were insignificant in comparison to the assets of "The Telephone". However as more and more corporations expended greater and greater funds, revenues resulting from UNIX became "noticeable".

The increasing awareness of UNIX can be seen if we look at how it influenced the naming of organizations within Bell Labs. (Figure 2) This is quite a contrast to the academic UNIX, planned, developed, integrated and supported by all of two people with a bit of help from their friends. A bulletin board message in an obscure New Jersey Ivy League university refers to the recruiting of "regiments of myrmidons". They report to managers, committees and task forces directing the development of UNIX. Their purposes are varied and their priorities are often incompatible. It seems that UNIX is attempting to fill a role as a computing panacea. The modest but powerful servant has developed a megalomaniacal personality. And naturally, it's becoming paranoid too. The clever ideas that go into UNIX are being guarded rather than shared. Its development is marked by competition rather than cooperation.

New UNIX systems are continually appearing in the marketplace. Many serve only as a disappointing reminder of what UNIX once was. The few experiences we've had with modern versions of UNIX from various suppliers have been utterly frustrating. Almost every facility we used fell apart at the slightest touch. Support was either non-existent, not helpful, or not timely. This in itself was not unsurmountable, but coupled with the lack of source code, and no user community, it was fatal. We're troubled by the practice of unbundling UNIX. Once integral parts are fast becoming "options" (figure 3). One has to be careful to order all the pieces or one could wind up with nothing but bits. Imagine buying a brand new Thunderbird, and having to separately order the engine, and a wiring harness, and wheels, etc. and finding that the transmission for this model isn't quite working yet. "I'm sorry sir but you can only drive in reverse until 1st quarter 1985. However, we do have a model with overdrive in beta test." As more and more versions of UNIX appear, the value of is diluted. The community is becoming fragmented. At one time every user was a member of the same elite group. Now there are different sects, each with various orders and classes. We already suffer from the proliferation of really only two distinct UNIX versions. What will it be like when there are twenty?

Some of the main attributes of academic UNIX have paved the way to its downfall. Perhaps this is the "Peter Principle of Technology". Elegant simplicity has succumbed to complex efficiencies. Basic general designs were breeding grounds for amazing universal solutions. Portability led to incompatibility. The lakes, forests and minerals of UNIX have been polluted, cut down and strip mined.

So what? Isn't that what resources are for? Isn't that the natural course of events? We suppose it is, to some extent. But we can progress in a more rational manner. We can take advantage

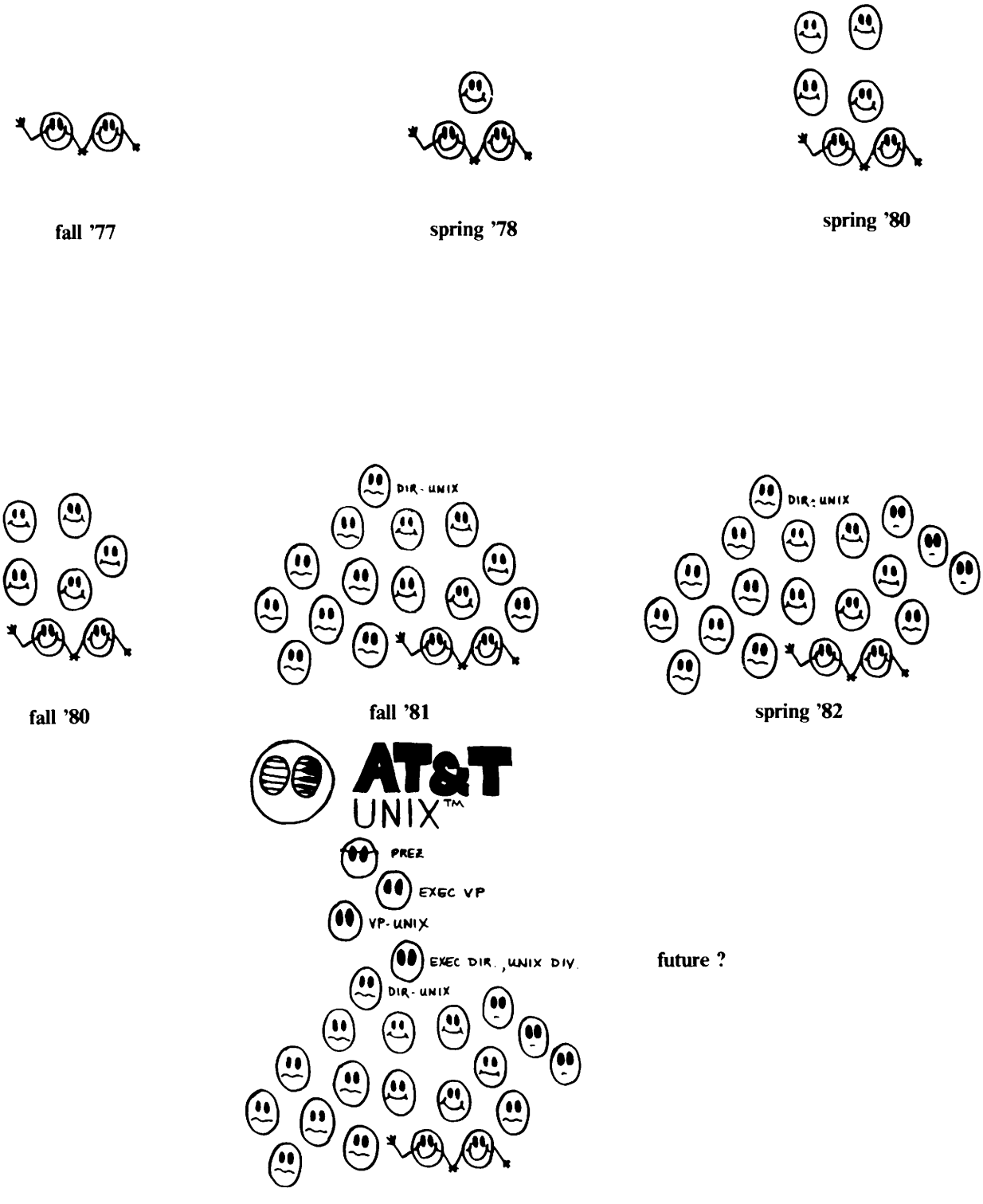


Figure 2.

of the lessons of academic UNIX and still preserve its essence. It's not too late to start making the distinction between the academic UNIX that was designed to provide a fertile environment for computing research and the commercial UNIX that has grown out of it to support any number of specific applications. Perhaps academic UNIX can be preserved if it is viewed as a legitimate application itself. Ergonomic designs don't seem to take our needs into account. We don't want a "user-

UNIX PARTS LIST

PART	PRICE	DESCRIPTION
/UNIX	$ 800.00	BOOTABLE IMAGE
/BIN	NO CHARGE	COMMAND DIRECTORY
/BIN/SH	$ 100.00	BASIC SHELL
/BIN/CSH	$1200.00	BERKELEY SHELL
/BIN/KSH	$1850.00	JACK-OF-ALL-SHELLS
/BIN/LS	$ 25.00	OPTIONS SOLD SEPARATELY, $5.00 EACH
/BIN/CAT	$ 3.00	CAUTION: THE PROGRAMMER GENERAL HAS DETERMINED THAT THE -V OPTION MAY BE HARMFUL
/BIN/ECHO	WRITE FOR PRICE	

Figure 3. Options

friendly system". We are not friendly users and neither are our colleagues. We're inconsiderate ogres without the slightest regard for the machine. We expect it to respond on command, to work endlessly and not to put up a fuss, rather like a mute slave whose only purpose is silently to obey.

As another example, the UNIX that is geared for a hostile environment (like industry) is not

appropriate for our needs. Where it may be perfectly reasonable to protect users from each other by ensuring that their files are unreadable, that's nothing more than a road block to the sharing of information required in a research environment.

As UNIX itself is a product and is molded to meet the needs of a diversified consumer market, it's time to reconsider its development. It's inappropriate for it to be both an end product for the computing consumer and a base for further applications. Another UNIX must be made available to provide a sound foundation. Academic UNIX provides the model for such a system. That UNIX should be available with source, with support, and on various types of hardware. It should be considered THE certified UNIX upon which all others are based. Its development will be thoughtfully administered by a small group of dedicated monks. Perhaps that is how we can have our cake and eat it, too.

We've strayed from our abstract, but let us put it right by saying that we believe that UNIX is entering a period of high visibility and enormous growth. There's no doubt in our minds that UNIX will be the standard operating system for personal computers in this decade. Although it's clear that academic UNIX will not be appropriate for the majority of personal computing users, whatever form UNIX takes will surely be influenced by the good ideas that went into its original design. And although Jack and Jill Hacker may have no idea what UNIX was all about or what it stood for, they'll undoubtedly be subtly influenced by its underlying principles.

Epilogue: There is a subtle non-technical aspect of UNIX past. That is, as a small system, UNIX was able to be less harsh, less uniform, less antiseptic than is customary or businesslike. UNIX was a revolutionary system. It rebelled against traditional views of the responsibilities of the system and the user. UNIX was smug, irreverent, cliquish and sarcastic.

Society (computing society) has laid a burden upon UNIX. It is looking to UNIX to fulfill the promises inherent in its design and philosophy. As UNIX accepts this responsibility, it conforms to other expectations. The brash, irreverent, radical attitudes that pervade it give way to stability, clarity and uniformity. Such attributes were not necessary in the academic UNIX. In fact their absence (or lack of emphasis, to be polite) contributed to a colorful and interesting environment. We've heard that paradise is boring!

In light of IBM's recent announcement, we observe that UNIX has donned a suit and now fits in with its new surroundings. We hope that some of UNIX's personality will rub off on its more traditional associates. UNIX isn't going to revolutionize IBM, but its adoption by IBM is evidence of its influence. Furthermore, in light of AT&T's recent announcements of new processors, terminals and assorted hardware, we see that UNIX has become quite important. Perhaps it has become too important to serve our modest needs.

References

1. Ritchie, D. M., and Thompson, K., "The UNIX Time-Sharing System", *Comm. ACM*, **17**, 7, July 1974, pp. 365-375.
2. Kernighan, B. W. and Plauger, P. J., *Elements of Progrmming Style*, McGraw-Hill, New York, 1974.
3. Kernighan, B. W. and Plauger, P. J., *Software Tools*, Addison-Wesley, Reading, Massachusetts, 1976.
4. Kernighan, B. W. and Ritchie, D. M., *The C Programming language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
5. Kernighan, B. W., and Mashey, J. R., "The UNIX Programming Environment", *Software Practice & Experience* **9**, 1, January 1979.
6. Kernighan, B. W., and Pike, R., *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

7. Ritchie, D. M., Turing Lecture, 1983 ACM Annual Conference, October 24, 1983. To appear in *Comm. ACM.*

Measuring Disk I/O on a VAX

Zdravko Podolski
Nick Nei

University of Glasgow
Computing Science Department
14 Lilybank Gardens
Glasgow
Scotland

(zp@glasgow.UUCP, i.nei@glasgow.UUCP)

ABSTRACT

Measurements and their statistical analysis enable phenomena to be treated quantitatively. This paper describes a project underway at the University of Glasgow to gather statistics on disk performance on a VAX running Berkeley UNIX 4.1BSD. A technique to conduct highly detailed and accurate measurements with negligible perturbation to system performance is described. Indeed this powerful method is applicable to the measurement of any part of the kernel.

A brief survey of past work in description and analysis of stochastic computer processes and computer communications traffic is presented. The rationale and the method of measurement of the project are explained, followed by an overview of the results. An analysis of these results is also provided, especially against the previous common assumption of an exponential distribution for interarrival times. That this stochastic process can be closely approximated by the Poisson process is considered crude, and possibly invalid, particularly in the light of current advances in computer and computing technology, and of a unique, popular and modern operating system like UNIX. The customary queuing theory model so frequently used in performance studies is therefore rendered impotent in this context.

Finally, possible directions in attempting to characterise the variables of the empirical model are suggested.

1. INTRODUCTION

Many researchers have found intercommunication traffic in computer systems to be quite close to the Poisson process. A rêsumê of their work follows.

Over 80 years ago, the studies of Erlang [1] to characterise toll telephone system behaviour resulted in the Poisson arrival process and the exponential interarrival time distribution. This inspired the work of Fuch and Jackson [2] more than 10 years ago, in computer communications traffic. They showed that the exponential distribution closely approximated the random variables of the empirical estimates, and noted with interest that these characterisations retained their validity throughout the years despite the many technological changes. There is also the work of Anderson and Sargent [3] who in their research in performance variables tested and confirmed that interarrival times were nearly exponentially distributed. Maisel and Gnugnuli [4] also confirmed an exponential distribution of interarrivals times of terminal usage at social security district offices too. And finally, the recent work of O'Neill and O'Neill [5] to characterise input traffic on several interactive facilities also found that interarrival times could best be approximated by a gamma distribution, while the connect, inactive CPU, and active CPU times can be approximated by exponential distributions.

Efforts have been underway at the Department of Computing Science of the University of Glasgow to measure and analyse the arrival process of requests for disk I/O of a VAX 11/780 running under Berkeley UNIX 4.1. Such a project has several interesting facets. One of them can be

running under Berkeley UNIX 4.1. Such a project has several interesting facets. One of them can be attributed to the popularity of UNIX and its enviable position of fast becoming the de facto standard operating system of today. This is particularly so in the light of the comment by Fuch and Jackson on the validity of their results despite years of technological development. The other is due to the uniqueness of UNIX. Unlike other operating systems, UNIX in the words of its original designer, Thompson, "is an I/O multiplexer more than a complete operating system". [6] Thompson's own interests after the Multics debacle was to build a file system rather than an operating system.

The rationale and the method of measurement is first described. Then the results of a series of measurements are presented. Finally, an analysis of these results are presented.

2. WHY MEASURE?

Measurement allows phenomena to be treated quantitatively. The customary measures of spread and central tendency - i.e. the mean and the variance, are useful indices of the shape of the distribution. The coefficient of variation, defined as the ratio between the mean and the standard deviation, can be employed to determine the dispersion of data. The coefficient of variation of an exponential distribution is known to be unity. The following criteria can be used to test the interarrival characteristic of I/O traffic: [7]

$$\begin{aligned} 0 &< C < 0.7 &&: \text{evenly-spaced arrivals} \\ 0.7 &\leq C \leq 1.3 &&: \text{Poissonian arrivals} \\ 1.3 &< C &&: \text{clustered arrivals} \end{aligned}$$

where C is the coefficient of variation.

In the realm of performance evaluation, measurement can be regarded as the most important evaluation technique. The evaluator has at his disposal modeling techniques as well, but in the final analysis the fidelity of the model can only be verified against measurements. [8] It is in the employment of modeling that measurement is appreciated. The behaviour in time of a model, under stimuli which represents the system's environment, mimics the system itself. The optimisation of time-shared system performance requires the description of the stochastic processes governing the system activity. Very often, such measurements and their evaluation provide an understanding not obtainable from queuing theory models. [3] A statistical description of such activity therefore becomes an invaluable tool for system reconfiguration forecasts, system tuning, device selection studies and etc.

3. WHAT TO MEASURE?

The interarrival times of requests for disk I/O transfers are defined as the intervals between successive arrivals of such requests.

The VAX in the Department is configured with two RM03 disks both attached to the Massbus Adapter (MBA) and interfaced by a disk driver routine called *hpstrategy*. Figure 1 illustrates.

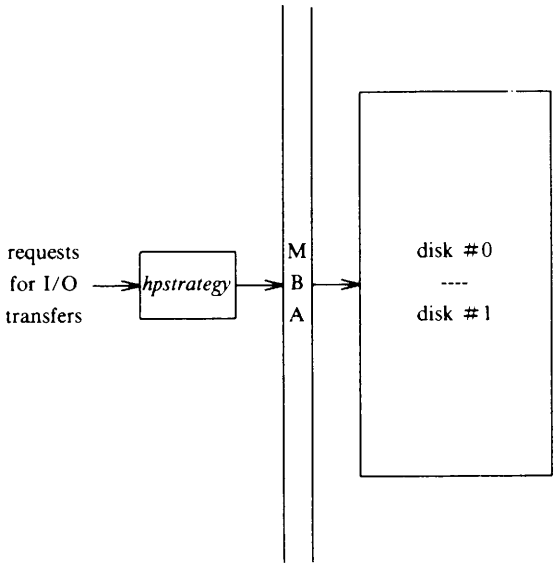


Figure 1 : Logical Configuration of the Disk I/O subsystem

The two disks can be conveniently regarded logically as a single service-centre. Hence, the most expedient point of measurement is at *hpstrategy*. This routine can then be instrumented to detect and measure interarrival times.

The choice of *hpstrategy* is not the only possible one for disk subsystem instrumentation. One could also measure interarrival times at, for example, the disk dispatch routine *hpstart,* or indeed at the system call interface (the various system calls which initiate I/O). Each set of measurement would convey different meaning, and future work will attempt to determine the relationship between them.

4. HOW TO MEASURE?

The customary way of organising data derived from observations is to present them as a frequency distribution or histogram. The most expedient implementation is to represent the histogram as an array of integers which count the number of times a variable falls in different intervals or cells. Each measured interarrival time then is modulus divided by the size of the array, and the result used as a subscript to index the appropriate integer to be incremented.

An array of 500 integers was used to implement this histogram. This technique exploits the large memory capacity of the VAX, and as will be discussed below, presents a time-space trade-off which can be comfortably balanced.

This approach however, has certain implications. Because the size of the array is fixed, any large interarrival time outwith the range of the array will be excluded from the histogram. Distribution truncation will result. This is not necessarily bad. In fact, the range of interarrival times is so large that an asymptotic distribution can be observed. The asymptote comprises large and infrequently occurring values commonly known as "outliers". These carry a large weight when estimating the mean and the variance of the distribution and therefore distort these statistics. Truncation is therefore even recommended, and will have negligible effect on the resultant function that characterises the distribution. If the negative exponential distribution function is expressed as

$$f(t) = 1 - e^{-\lambda t}$$

where f(t) is the probability that the interarrival time is less than t and where λ is the mean arrival rate of requests to the disk subsystem, then the distribution truncated to the right at $t = \gamma$ microseconds would have the following probability density function:

$$f'(t) = 1 - \xi e^{-\lambda t} \qquad \text{if } t < \gamma$$
$$= 0 \qquad \text{if } t \geqslant \gamma$$

where ξ is determined from the condition that

$$\int_0^\infty f(t)dt = 1$$

and thus

$$\xi = 1 - e^{-\lambda\gamma}$$

Since the magnitude of γ is extremely large, (γ for example, can be in the region of 250,000 microseconds), ξ can be ignored since it approaches 1 and therefore has negligible effect on the function.

Hence

$$f'(t) \approx 1 - e^{-\lambda t} = f(t)$$

In fact, this technique of truncation constitutes a "laundering" of contaminating outliers from the data.

It will be noted that both parameters λ and γ are specified a posteriori. γ is determined by the interval width and the array size, and therefore also determines the degree of truncation. The investigator cannot easily vary the array size, but can specify the interval width for any measurement session. This regulation of the interval width is most conveniently implemented by the use of the I/O control command *ioctl* (2).† This enables the interval size to be set in the kernel for different measurements without the need to reboot the operating system each time.

Hence, a certain amount of control of γ (and hence the degree of truncation) is possible. Table 1 shows the correlation between the interval width and the degree of truncation for several measurements, and serves as a useful guide. † Round parenthesised numbers refer to section numbers of the UNIX Programmer's Manual.

A trade-off between the size of the interval and the degree of truncation therefore results. The investigator has to make a judicious choice to compromise this trade-off against the amount of laundering.

Interval Size (microseconds)	Degree of truncation (%)
1	99.99
5	84.98
10	87.84
25	67.35
50	46.59
75	36.24
100	24.75
250	9.39
500	2.62
750	1.50
1000	1.40
2000	0.48
5000	0.59
100000	0.00

Table 1 : The Trade-off between Interval Size and Degree of Truncation.

This varying of the interval size can be advantageously exploited. When the interval size is too large, the histogram will be too granular, and detail will be lacking. To achieve high resolution

in the distribution, the interval size needs to be sufficiently small. With this in mind, several measurements were taken at decreasing interval sizes. This control of the granularity of the measurements can be likened to the use of a zoom lens in photography to bring pictures from a distance to a close-up without moving the camera. The smaller the interval size, the finer the grain and the greater the detail.

Finally, consider the time resolution of the measurements. A high resolution in microseconds not only helps achieve precision and accuracy but also enables time to be treated discretely. This greatly facilitates arithmetical work since most computations will avoid floating-point arithmetic which can significantly perturb the system. Unfortunately, the UNIX kernel tells time at millisecond resolution. To tell time at microsecond resolution the VAX Interval Count Register (ICR), which ticks at every microsecond is used. The question which confronts the investigator is: will it take too much time to tell time? The assertion of this work is that such perturbation to the system and the measurements is negligible. This is discussed below. Indeed, the elegance of this technique with its minimal perturbation and yet high resolution makes it an attractive method for measuring any part of the kernel.

5. RESULTS OF THE MEASUREMENTS

Seventeen measurements at different interval sizes were taken, but for brevity, only 8 frequency distributions will be examined. All the histograms are expressed as relative frequency distributions, all values being percentages of total interarrivals (i.e., regardless of the truncation). First some general observations.

A visual examination of the distributions suggests that they are exponentially distributed. Observations display a general clustering of values near the y-axis and a long-tailed asymptote in the x-direction. The data appear to justify satisfactorily the common assumption that the interarrival times are serially independent. Based on this, as a good approximation one may assume that, for an average disk sub-system, the length of any given interarrival period is statistically independent of the lengths of all previous periods. But, as will be considered later, the feature of long serial reads in the I/O system may refute this conclusion.

The average time taken to process an interarrival time - i.e. to tell time, increment the histogram and dynamically estimate the mean and variance, is in the order of magnitude of 0.74 milliseconds. Benchmarks were used in order to assess the degree of perturbation this causes to the system. These were timed by the *time* (1) facility of UNIX which gives the elapsed time during the benchmark (real), the time it spent in the system (sys), and the time spent in execution of the command (user). Five benchmarks were taken without the measurement, and then five were taken while measurements were being conducted and analysed. The averages are calculated, and summarised in Table 2.

No Measurement		
real	user	sys
257.5	220.00	16.82
With Measurement		
real	user	sys
304.5	217.25	17.46

(in seconds)

Table 2 : Benchmark Results.

The variation between the means were analysed using ANOVA (Analysis of Variance). For all three timings, the null hypothesis that there is no difference between the benchmarks could not be rejected at 5% level of significance. It can therefore be concluded that the measurements themselves do not significantly perturb the system, and with the large memory capacity of the VAX, the extra memory used by the instrumentation in the kernel code does not noticeably affect system performance.

Figures 2 to 9 present the relative frequencies of 8 measurements taken at various intervals. Table 3 serves as a useful index to these figures.

Figure no.	Interval size
2	100 millisecs.
3	2 millisecs.
4	1 millisecs.
5	500 microsecs.
6	100 microsecs.
7	50 microsecs.
8	25 microsecs.
9	5 microsecs.

Table 3 : Frequency Distribution Index.

The measurement strategy renders a zoom effect, and this is how the distributions should be surveyed. For example, Figure 4 is a magnification of Figure 3 up to approximately 200 milliseconds on the x-axis. Not the entire length of the 500-integer histogram is graphed in these figures. In fact, the full length will extend two and a half times more. This gives an idea of the expanse of the range and the severe asymptotic character of these distributions. A brief survey follows.

At a coarse resolution of 100 milliseconds, two clusterings can be observed, one close to 0 microseconds, and the more curious one clustering around 30 seconds. This is explained by a program called *update* (8), which is executed every 30 seconds and does a *sync* (2) system call to flush out all disk buffers, so that disks would never be more than 30 seconds behind the in-core buffers. These should clearly be laundered during mean and variance estimation as prescribed above.

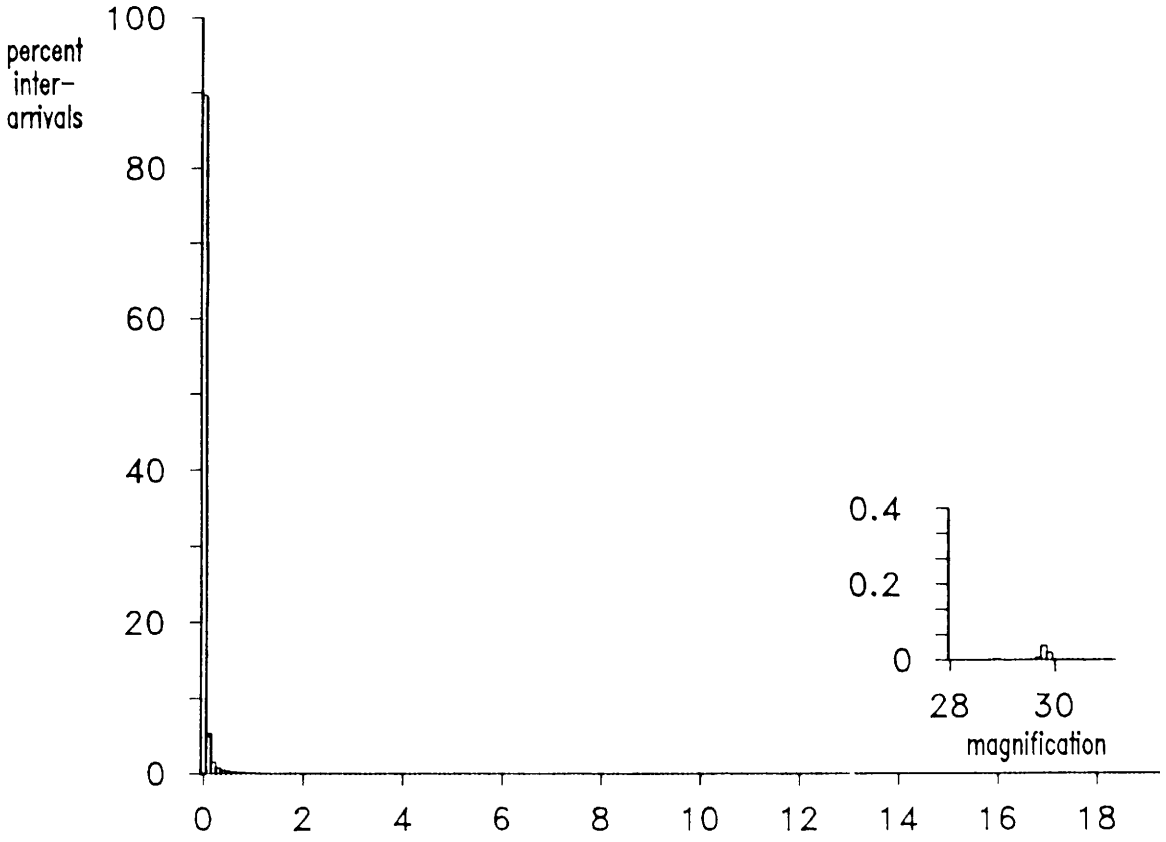


Figure 2. Measured at 100 Millisecs. Interval

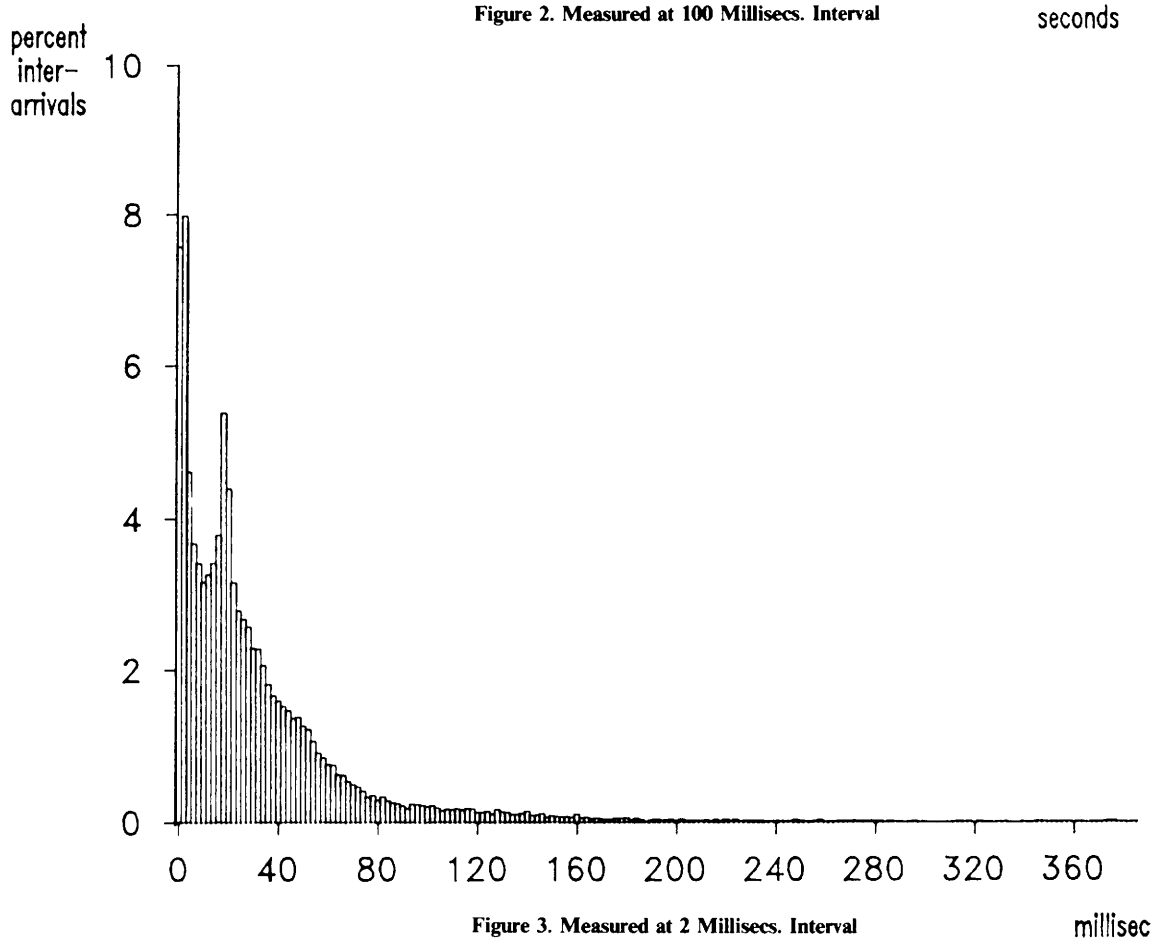


Figure 3. Measured at 2 Millisecs. Interval

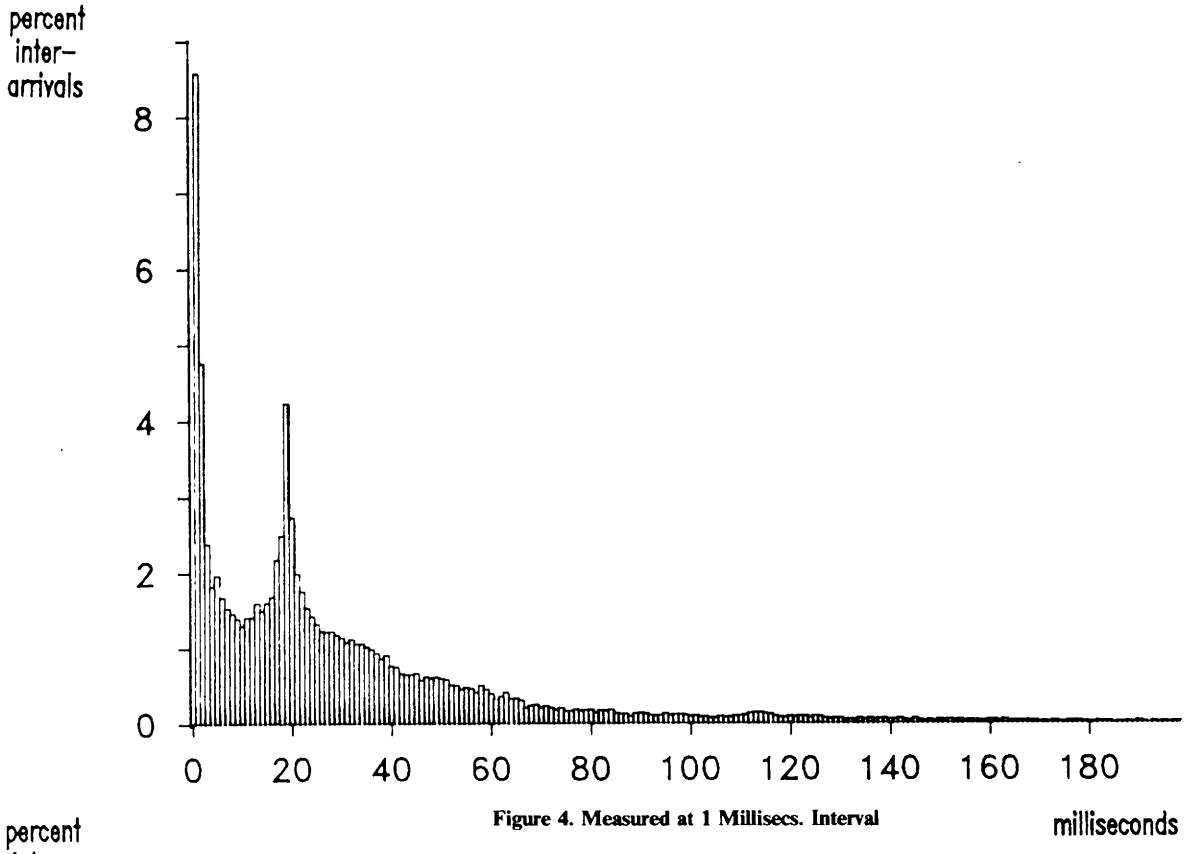


Figure 4. Measured at 1 Millisecs. Interval

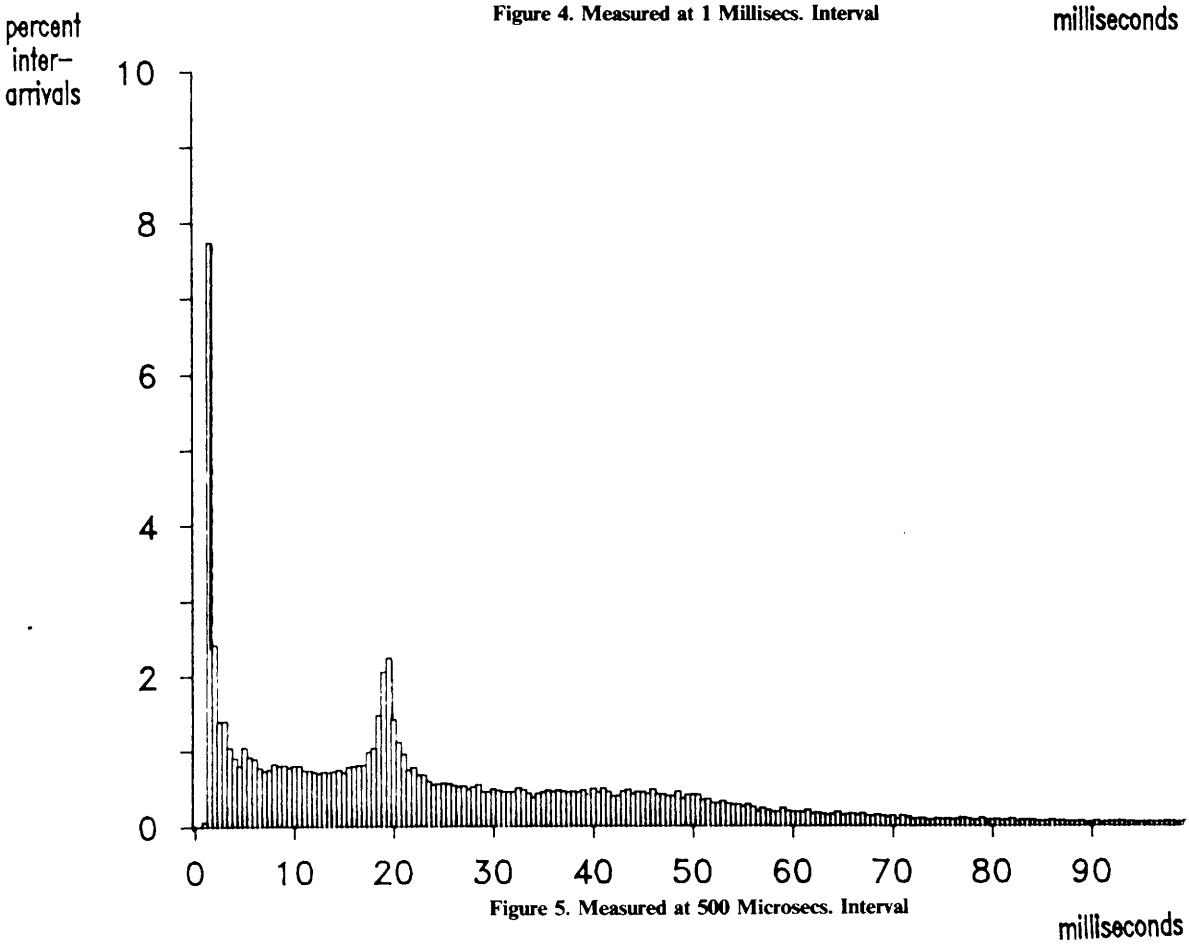


Figure 5. Measured at 500 Microsecs. Interval

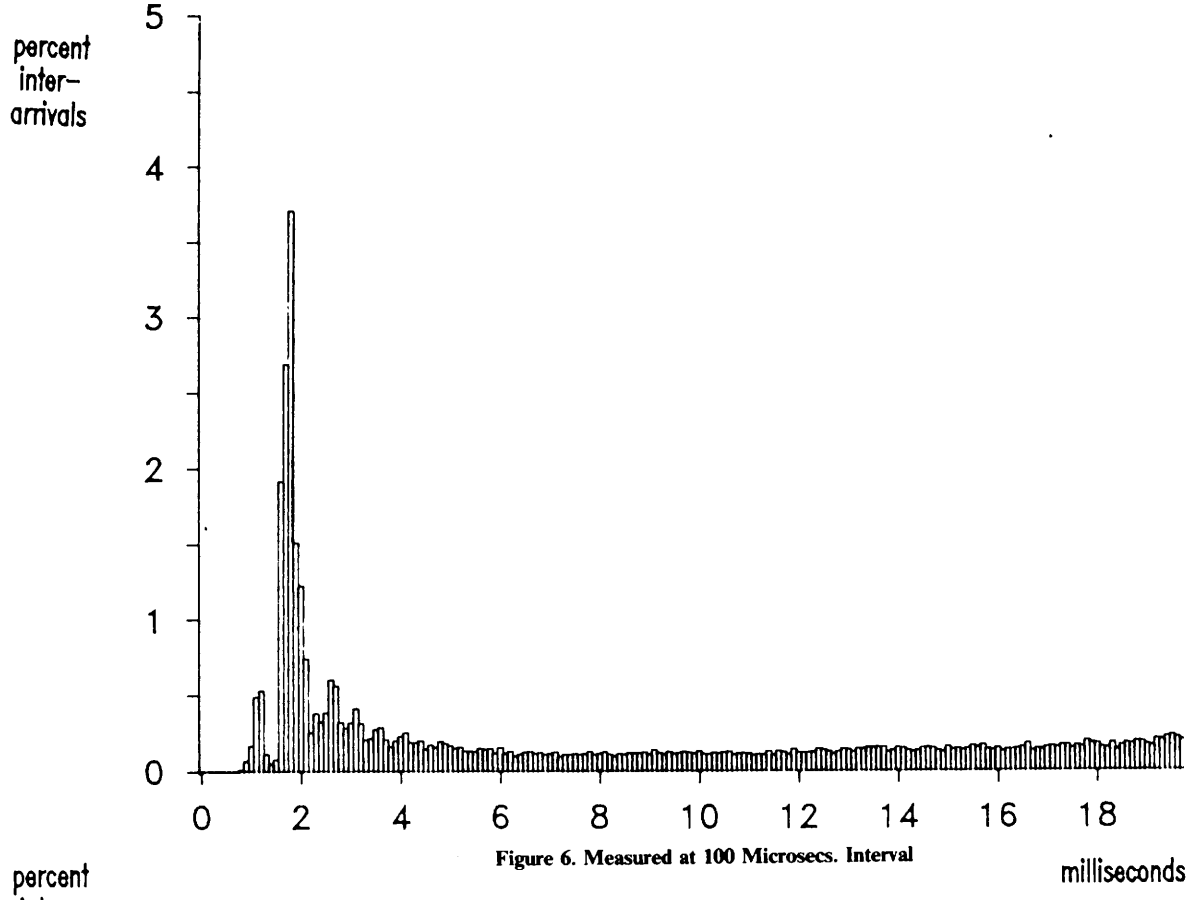


Figure 6. Measured at 100 Microsecs. Interval

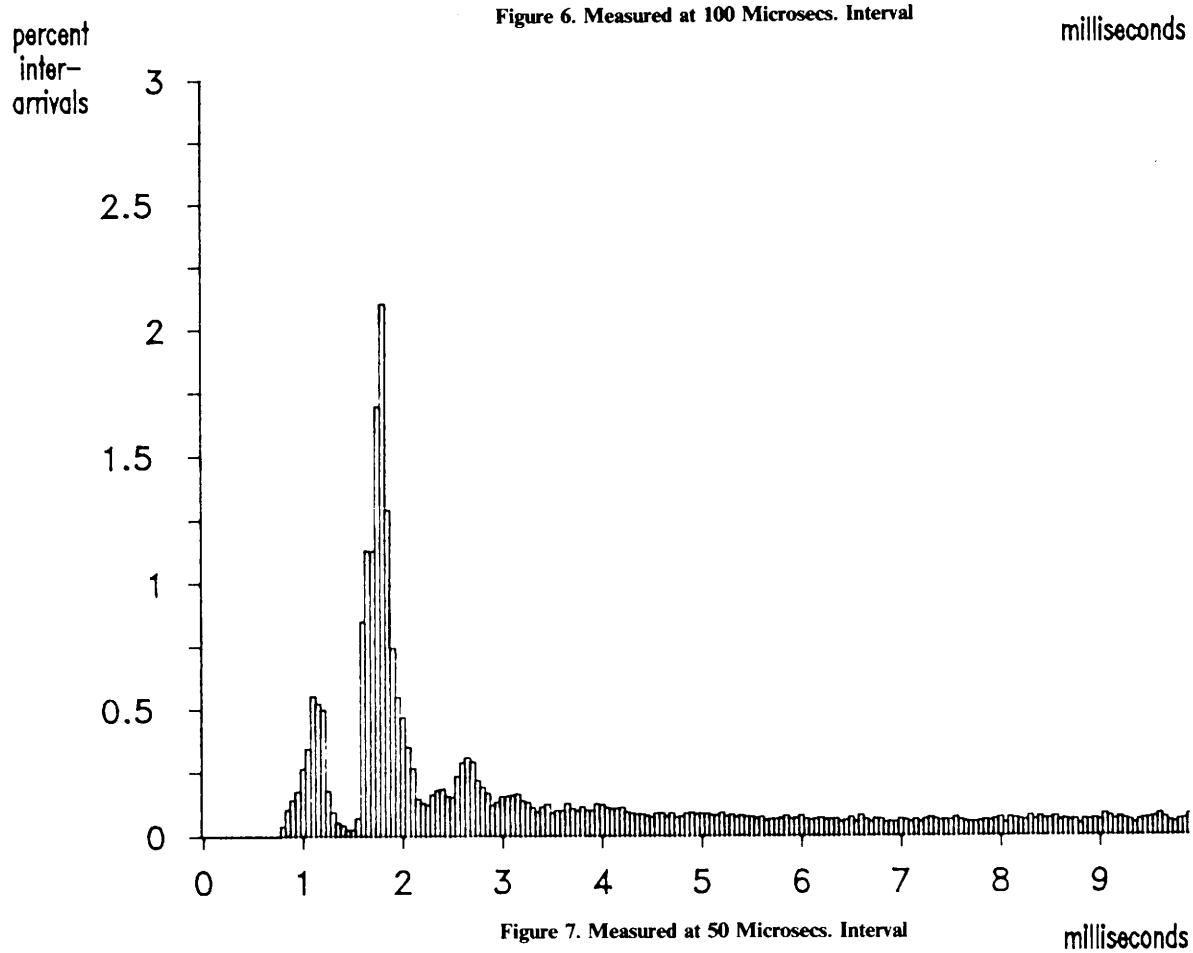


Figure 7. Measured at 50 Microsecs. Interval

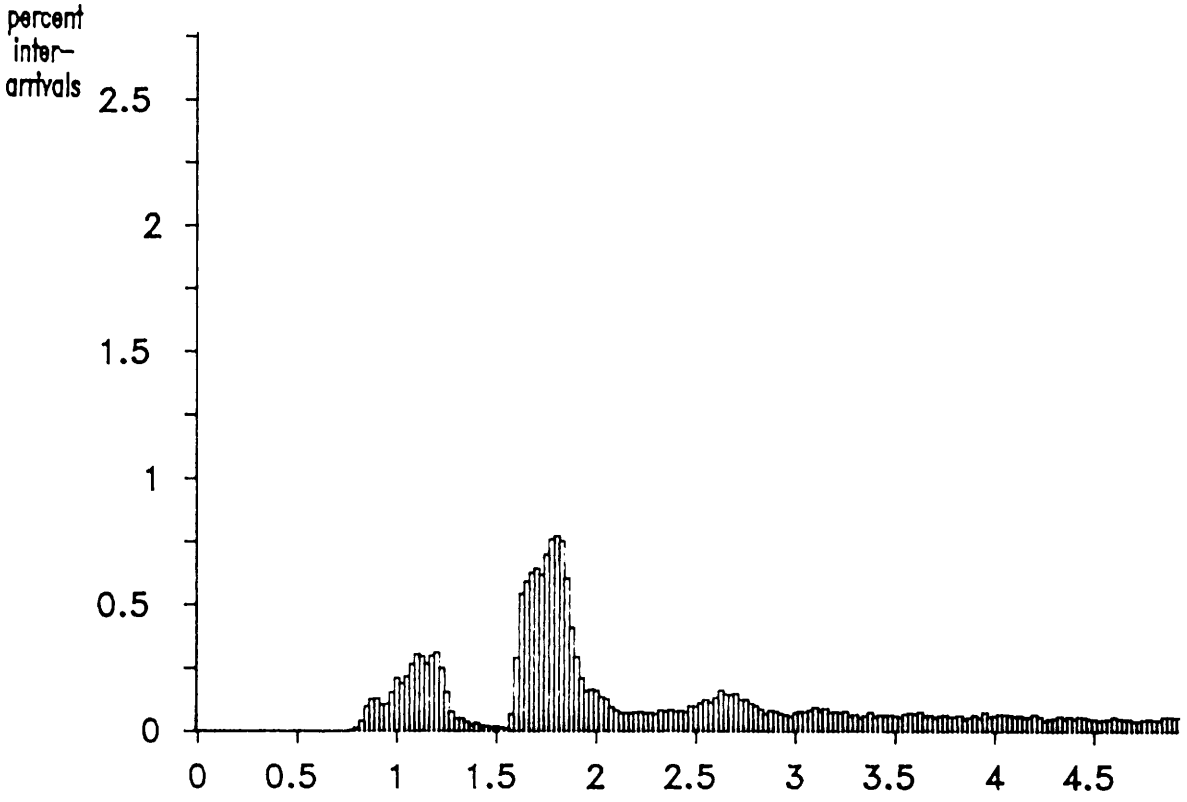


Figure 8. Measured at 25 Microsecs. Interval

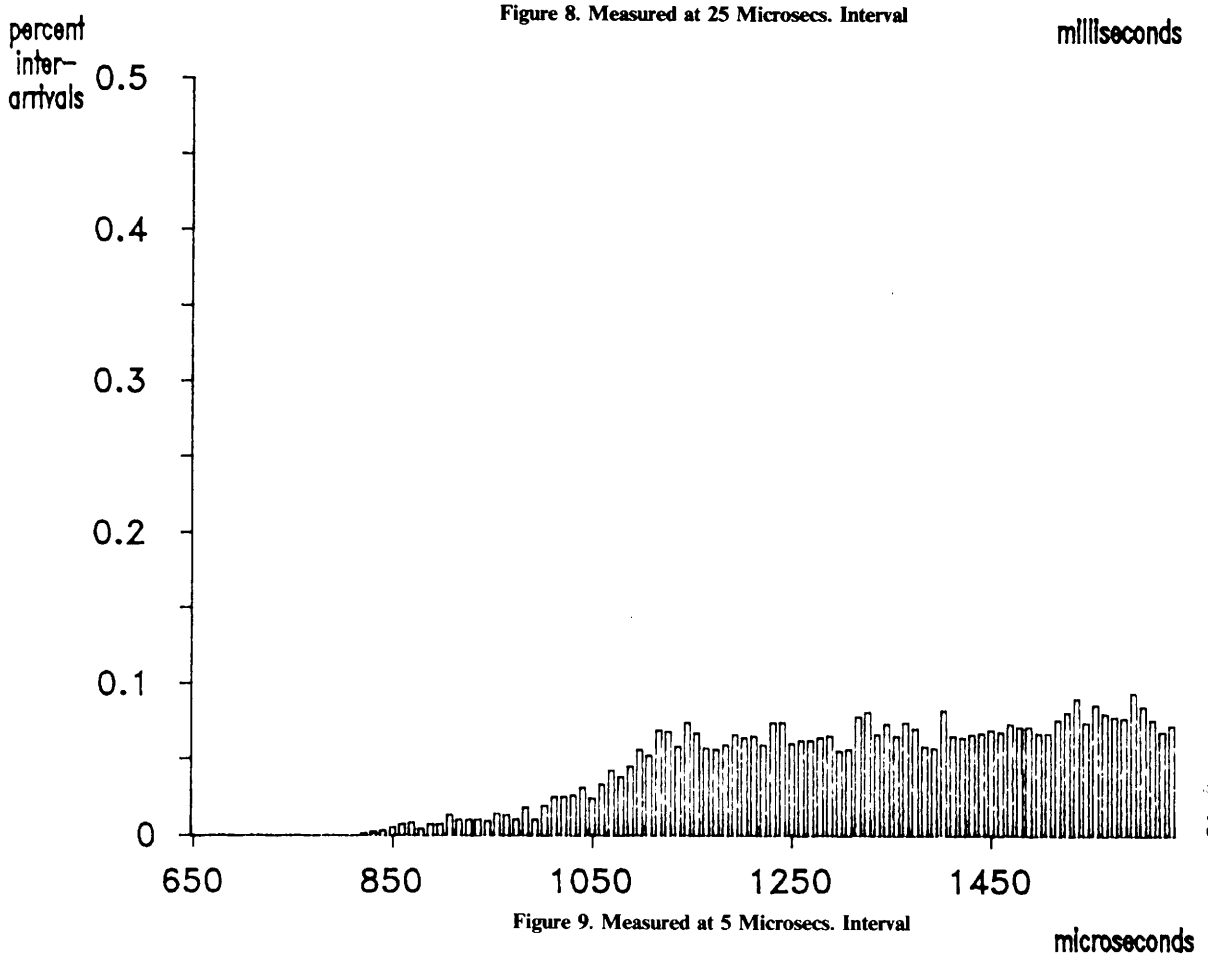


Figure 9. Measured at 5 Microsecs. Interval

A measurement at 2 milliseconds magnifies the other clustering, and in fact, another clustering can be observed at around 20 milliseconds. Further measurements at 1 millisecond and 500 microseconds reveal the shape of this clustering. It appears bell-shaped, as if normally distributed.

Measurements at narrower intervals at 100 and 50 microseconds zoom onto tne shorter interarrival times. Two clusterings now appear - at about 1 and at about 2 milliseconds. And Finally, Figures 8 and 9 show the degree of resolution at which measurements can be conducted. At the interval of 5 microseconds, the distribution show a clear absence of interarrival times less than 0.75 microseconds approximately. This is consistent with the 0.74 milliseconds overhead required to measure an arrival for disk I/O.

6. ANALYSIS OF THE RESULTS

Recall that the coefficient of variation is recommended as a crude indicator to the distribution type. Obviously, severely truncated and untruncated (and hence contaminated) distributions distort means and variances, and therefore also distort the coefficients of variation. With the laundering of outliers (i.e. a moderate truncation) to obtain a more rational mean and variance, many of the distributions exhibit coefficients of variation close to unity and within the Poisson-arrivals range of 0.7 and 1.3 discussed above, indicating Poisson arrivals. This reveals the inadequacy of the coefficient as an indicator since it is dependent on the degree of truncation. Figure 10 illustrates this correlation.

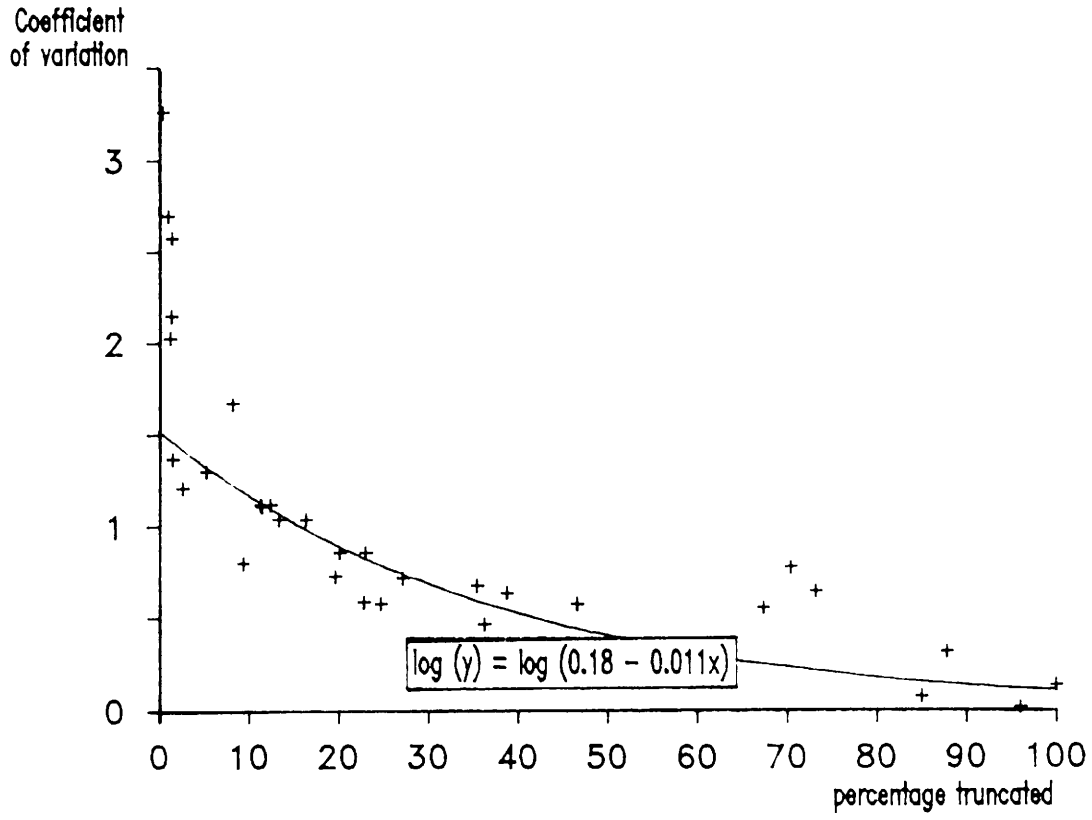


Figure 10. Regression of Coefficient of Variation vs. Degree of Truncation.

Attempts were made to fit the measured distribution to a negative exponential distribution. The measurement taken at 500 microseconds interval was chosen for its moderate truncation and because it is a fair population distribution archetype. Poisson arrivals were simulated and compared with the empirical distribution. The result is presented in Figure 11, where the histogram represents simulated interarrivals times, and the curve represents measured interarrival times.

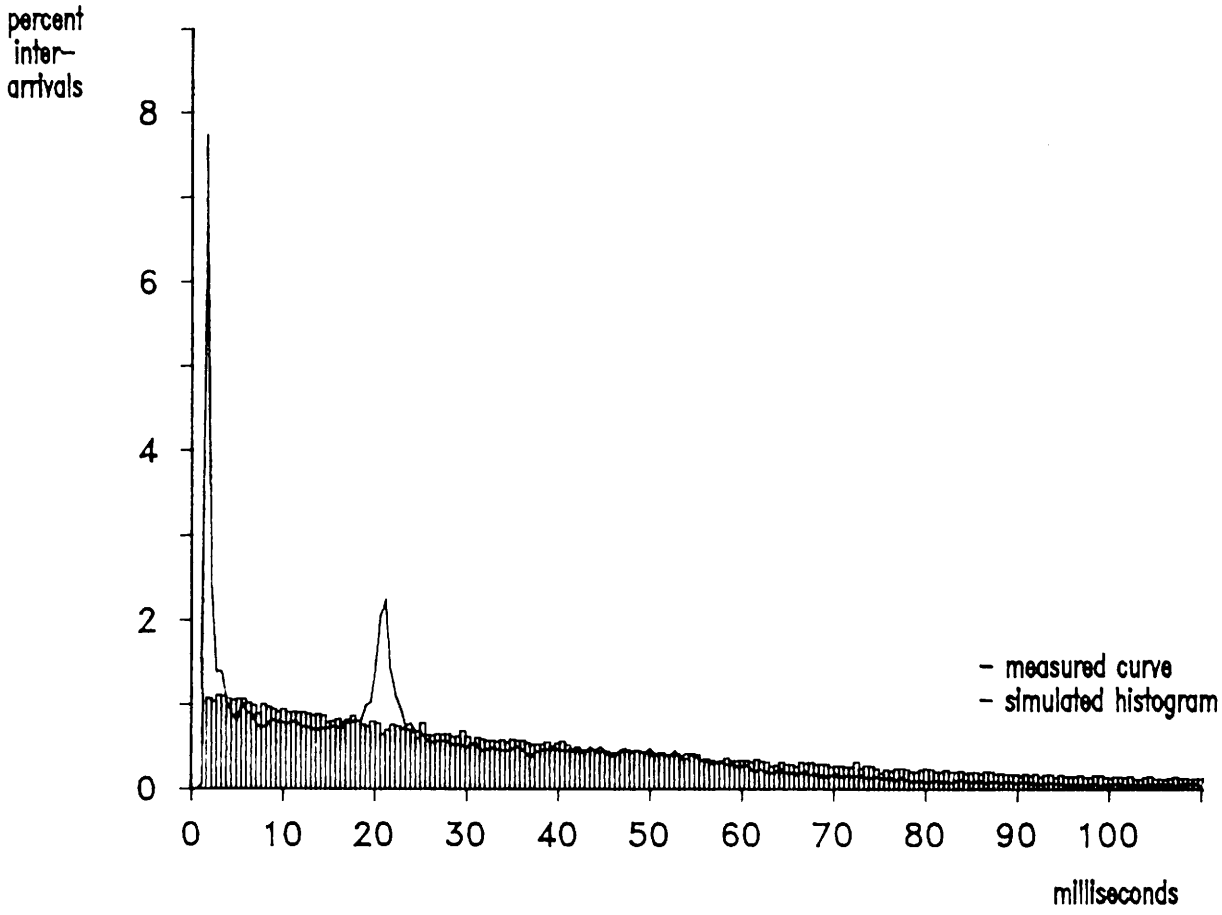


Figure 11. Fitting with an Exponential Distribution

It can be visually inferred that the distributions fail to match. The empirical distribution possesses far too many extreme values at the asymptote, and the exponential distribution for all purposes underestimates the distribution near the origin. More interestingly, the minor clustering of values around 20 milliseconds cannot be accounted for. The customary Chi-square and Kolmogorov-Smirnov tests for goodness-of-fit were applied. Both rigorous tests rejected the null hypothesis that the distributions were equal.

Another reasoning against the appropriateness of an exponential function to characterise interarrivals can be founded on the fact that it requires a finite amount of time between requests. This is especially true in certain forms of communication traffic. For example, Grenander and Tsao[9] argued that since it requires humans at least a tenth of a second to respond, shorter values of interarrivals are not likely. A similar characteristic can be clearly observed in some of the measurements. Figure 5, of a measurement taken at 500 microseconds, for example show a marked absence of interarrival times observed at less than 500 microseconds. Indeed, the proportion of interarrival times between half and 1 millisecond is negligible.† This fact conflicts with the exponential density which assigns the highest probability density to the smallest time interval of length zero. † Note that this is accounted for when fitting the exponential distribution to the measured distribution in Figure 11. The fit is better, but still poor.

Some investigators have suggested that a much more satisfactory approximation can be obtained with a hyperexponential distribution. Coffman and Wood in their work to describe user channel traffic in a time-sharing system found the usual assumption of an exponential distribution as only a very rough approximation. But by assuming the distribution to be a biphase,

hyperexponential distribution (i.e. by a linear combination of two ordinary exponential distributions), they found a very adequate approximation. [10]

This was attempted, but unfortunately goodness-of-fit tests rejected the hypothesis as well.

7. FURTHER WORK

The conclusion must be that the assumption of Poisson arrivals is a very crude approximation. Unfortunately, it is only for completely random (Poisson) and regular arrivals that general mathematical solutions have been obtained.

Obviously, the next immediate step in future work is to attempt to explain the anomaly. What gives rise to the shape of the distribution? Since the Poisson assumption is at the very best but crude, one may speculate as to whether the underlying distribution is indeed exponential or hyperexponential, but perhaps distorted by some feature which may be peculiar to UNIX. Several come to mind, not the least being the *read-ahead* feature and the *paging* features of UNIX. Both or either of these, can contribute to the minor clustering, whose shape and origin is still a matter of speculation. The examination of the *read-ahead* is intuitively attractive since the regularity at which it occurs would suggest that a proportion of interarrival time intervals are not entirely random - a characteristic not accounted for in applications of the Poisson distribution in queuing theory. This is also true in cases of long serial reads.

Another worthwhile area of study is the degree to which the interarrival time distribution is affected by the system clock. Since it runs at 60 Hz, many events can be expected to be driven by the clock, and hence one might expect a cluster of I/O activity around such an interval (17 milliseconds plus).

Also, as intimated before, investigations into the relationship between system calls and I/O can be expected to yield enlightening and interesting results. And finally, further work is required to measure other machines to corroborate and augment these findings.

References

1. A.K. Erlang, "Calcul des probabilitès et conversations telephoniques," *Revue Generale, de l'Electricitè* **18**(Aug. 1925).

2. E. Fuchs and P.E. Jackson, "Estimates of Distributions of Random Variables for Certain Computer Communications Traffic Models," *Communications of the ACM* **13**(12) pp. 752-757 (Dec. 1970).

3. H.A. Anderson and R.G. Sargent, "A Statistical Evaluation of the Scheduler of an Experimental Interactive Computing System," pp. 73-98 in *Statistical Computer Performance Evaluation*, ed. W. Freiberger, Academic Press, New York (1972).

4. H. Maisel and G. Gnugnuli, *Simulation of Discrete Stochastic Systems*, Science Research Associates, Chicago (1972).

5. P. O'Neill and A. O'Neill, "Performance Statistics of a Time Sharing Network at a Small University," *Communications of the ACM* **23**(1) pp. 10-13 (Jan. 1980).

6. K. Thompson, "UNIX Time-Sharing System: UNIX Implementation," *Bell Sys. Tech. J.* **57**(6) pp. 1931-1946 (1978).

7. IBM, *Data Processing Techniques: Analysis of Some Queuing Models in Real-Time Systems*, IBM Publications Dept., New York (2nd Ed. Sept 1972).

8. D. Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall Inc, New Jersey (1978).

9. U. Grenander and R.F. Tsao, "Quantitative Methods for Evaluating Computer System Performance: A Review and Proposals," pp. 3-23 in *Statistical Computer Performance Evaluation*, ed. W. Freiberger, Academic Press, New York (1972).

10. E. G. Coffman and R. C. Wood, "Interarrival Statistics for Time Sharing Systems," *Communications of the ACM* **9**(7) pp. 500-503 (Jul. 1966).

OSx: Towards a Single UNIX System for Superminis

Ross Bott

Pyramid Technology Corporation
1295 Charlseton Rd.
Mountain View, CA 94043
USA

(bott@pyramid.UUCP)

ABSTRACT

OSx is a dual port of 4.2BSD and System V onto the Pyramid 90x computer, a high end supermini. OSx is designed to be fully compatible with both 4.2BSD and System V in a fashion that neither suffers performance penalties from the coexistence of the other. This paper discusses some of the details of this design, both internal to the kernel and at the user interface level, along with some of the problems we faced in its implementation.

1. UNIX AT THE CROSSROAD OF SYSTEM V AND 4.2BSD

The direction of Unix on large machines is at a crossroads: two major Unix implementations exist, both with distinct advantages and disadvantages: 4.2BSD offers the virtual memory and disk I/O performance necessary to run Unix effectively on a large machine. System V provides the tools to run Unix within commercial environments, and, with the force of Bell Laboratories and Western Electric behind it, should provide better support and possibly better potential for future development. On the other hand, the large majority of high end superminis continue to run 4.1BSD or 4.2BSDbecause of its performance and features, and because the applications 4.1BSD users have been building for years were designed for this environment.

It is fairly likely that at some point these two systems will evolve towards a single standard. It is even more likely that this will take several years. During this period of dual standards, we feel that there need not be a barrier between the users of the two systems because of the incompatibilities. The OSx project had several design goals:

- To provide a system which could run equally well in either world, allowing installations to take advantage of the unique features provided by applications from both environments, as well as moving transparently between environments.
- As networked distributed operating environments become more prevalent, it will be a common occurrence that some machines on a network want to run System V and some 4.2BSD. OSx should provide a gateway by which these machines can coexist and communicate: Users on the network expecting either environment should get the features and compatibility they expect when communicating to OSx.
- Applications developers can use one environment while developing and testing their application to be fully compatible in the other world.
- For installations choosing to make a transition from 4.2BSD to System V or in the other direction, OSx should be a vehicle which can allow this to occur with as little pain as possible.
- OSx should be designed to allow smooth tracking of new releases from Western Electric and Berkeley, remaining compatible with the latest from either.

In addition, we believe that it is an important design constraint that clean interfaces be provided to both 4.2BSD and System V: The impulse to add a hodgepodge of extra features to either system should be resisted. This is particularly important for developers of applications on either system who want to be assured that their applications are portable to a native System V or 4.2BSD environment. It would do the UNIX world no favor to add yet another flavor of UNIX to an

already crowded field.

We have not taken a layered approach because neither system should have to pay a performance penalty. A large percentage of users are strict 4.2BSD or System V advocates, and aren't interested in having the opposite version coexist with theirs within the same operating system. A dual port is acceptable to these users only if there is no loss in efficiency to the features they are interested in. This is in addition to convincing them that no unanticipated incompatibilities were introduced by the necessity of supporting the opposite system.

Underneath the system interface in a dual port, both systems should be able to take advantage of whichever version provides the best performance and capabilities. For this reason, we are really a 4.2BSD internally:
We believe that 4.2BSD is clearly superior in performance, for reasons such as the fast file system and demand paging, to be discussed further later. We also feel that some of its features, particularly sockets, provide a useful general basis for other Unix services. (We are aware that Western Electric is "catching up", but high-end superminis can't afford to wait. On the other hand, if future releases of System V (or VI, etc.) contain internal enhancements which provide better performance, we won't hesitate to incorporate them.)

2. USER INTERFACE

We had several goals in trying to design an user interface for command execution and program development which would satisfy both 4.2BSD and System V users:

- The directory structure for command binaries, libraries, and header files should look exactly like what users of each system expect. For example, if a command normally exists in /usr/bin, it should still exist there. The decision to keep this standard isn't arbitrary: There are a considerable number of shell files and other applications programs which make explicit or implicit assumptions about the directory locations of programs (as well as a percentage of users with a very firm view of the directory structure).
- Commands, libraries, and headers which don't exist in a particular directory for a particular system should not be there. For example, 4.2BSD users should not see *dircmp* when listing /usr/bin, nor should they see regexp.h when listing /usr/include.
- If common commands have conflicting outputs or differing options, users of each type of system should have the outputs and options they expect, and only those. Thus, users of System V should be able to use the -ctime and -cpio options to the *find* command, missing from the Berkeley version. Conversely, 4.2BSD users should not see these options within their *find.*
- All of the above should be accomplished with no significant loss in system performance, and minimized impact on disk usage.

To accomplish the goals above is somewhat analogous to trying to get two different objects to occupy the same location at the same time. What we very specifically did NOT want to do was to "solve" the problem by introducing yet another UNIX version which contained an amalgamation of the features of both of the systems.

The solution we implemented was to introduce the notion of separate 4.2BSD and System V *universes,* coexisting in the same file structure and sharing a common kernel. At any given time, a user operates in one universe or the other. The initial universe is determined at login time by an entry in a /etc/u_passwd file. At any point the user can enter the alternate universe by typing one of the commands:

warp att (or just att) -- to enter System V

warp ucb (or just ucb) -- to enter 4.2BSD

The commands above actually fork a shell within the alternate universe, and using them is analogous to typing csh while using sh, e.g., one can return to the original universe by typing ^D, or continue to layer alternate universes. To determine in what universe one is operating, one can use:

```
% whereami
You are in the Berkeley 4.2BSD universe
%
```

If a user wishes to execute a single command in the other universe without entering it, this can be done by prefixing the universe, e.g, if one logged in as a System V user, typing:

```
ucb emacs
```

will allow a user to run EMACS within the ucb world. When the user exits, his System V world would be restored. Similarly, from a 4.2BSD world, typing

```
find . -name "*" -print | att cpio -oaB > /dev/rmt0
```

allows utilization of the System V-only cpio. Note that the inputs and outputs from commands from different universes can usually be piped to each other. Thus, in this case the *find* is a 4.2BSD version, with the output piped into a System V *cpio*. Wild card expansion and redirection for the whole command is done in the current universe, unless that portion of the command is quoted.

The concept of alternate universes is accomplished by implementing *conditional symbolic links.* These work like normal symbolic links, except that the directory or file to which the symbolic name points is dependent upon which universe the user is in. For example, to set up the /bin directory when the two universes are ucb and att, one would create two separate directories, e.g., /.ucbbin and /.attbin, then use the command:

```
ln -c ucb=/.ucbbin  att=/.attbin  /bin
```

Then, when one is in the 4.2BSD universe, /bin will look identical to /.ucbbin, both for executing commands and listing the directory. (By using periods before the names, these underlying directories will typically be invisible to users when listing directories.) We also added a new system call, *setuniverse,* for all users to set and switch the universe in which they are operating. (This system call is used by *ucb* and *att.)*

Up to five separate universes can be linked in the fashion above, although we currently plan to use it for 4.2BSD and System V universes only. These conditional symbolic links work as fast as symbolic links, and have negligible effect on performance. Their effect is further reduced by the command hashing within csh and the soon to be released System V Bourne shell.

We have used this concept for library and header directories as well as the common command directories.

3. SOFTWARE DEVELOPMENT INTERFACE

The concept of dual universes extends over into the program development area. There is considerable incompatibility in what System V and 4.2BSD programs expect out of the libraries and kernel interfaces, ranging from the meaning of the return value in printf to radically different mechanisms for handling signals. There are similar conflicts in the /usr/include and other directories of headers. The separate 4.2BSD and System V directories for libraries and headers allow us to completely maintain these distinctions. The right headers and libraries are automatically accessed by invoking the distinct cc (or f77, etc.) within the desired universe. The children of cc (e.g., cpp, ccom, etc.) will live within the same universe as that cc, and the header and library directories will be as viewed from that universe.

Once a program has been compiled, however, its behavior with respect to 4.2BSD or System V compatibility has been permanently established, and the program can be executed in either universe. Thus, the concept of universe must be distinguished from that of *application program environment.* The latter is a characteristic of each program running on top of the OSx kernel, and can be described by the following aspects:

- Program environments are NOT file system or process based. In contrast, a universe is a characteristic kept on a per process basis, and is used during file system access only to determine which underlying directory structure is relevant.
- The program environment of an object file is determined by the header files and libraries used to compile the application with.

- The kernel utilizes this program environment to provide differential 4.2BSD or System V behavior where they conflict.

One example of the distinction between universes and program environments can occur in application software development: If one is developing software for a particular version of UNIX, it is possible to "live" within the alternate universe while doing the development. For example, if a software developer prefers the tools provided by the 4.2BSD environment but is developing an applications package he wants to be transparently portable to a System V environment, then he can do all of his design and editing in a ucb universe, then compile his program using:

```
att cc -g -O SysVApplication
```

He could then debug his program using either of the following:

```
att sdb SysVApplication
dbx SysVApplication
```

Note that in the latter case, dbx runs within the ucb universe while the Sys V Application runs within the att universe. This is possible because, after the compilation, the flavors of system calls and header file assumptions made are all based upon System V standards. The kernel can then recognize that it needs to handle the System V system calls differently when necessary (details in the next section).

It was possible in some cases to modify 4.2BSD and System V code within libraries or headers such that neither interface is affected at all. However, we've taken the philosophy that, whenever there was even a remote possibility that combining the code would produce an unanticipated incompatibility within one of the universes, then separate versions should be maintained. There is the additional advantage that if and when new releases of System V or 4.2BSD are issued, then we can make a clean and fast transition to supporting the new versions.

The only exceptions to this philosophy are in areas which do not affect portability and allow jumping between universes without trouble. For example, we support a single object format (BSD style) and have full flexname symbols in either environment. This allows senarios such as the previous example, where debuggers from either universe can be used on a System V object file. (We will, however, offer a flag that can be passed to cc within the System V world which will complain about symbols which are not unique to the first 8 characters, so that applications developers can guarantee portability.)

4. DESIGN STRATEGY FOR THE DUAL PORT

Although the Unix operating system is by no means as modular as it might be, the kernel can be fairly cleanly divided into a set of concentric layers. The innermost layer concerns itself with virtual memory and process management and controlling of I/O devices, and interfaces with the architecture of the host machine. Ninety-five percent of the changes to port a Unix kernel to a given architecture are isolated to this layer. (In essense, this layer insulates the rest of the kernel from the host architecture.)

Conversely, the particular version of Unix interface supported (e.g., 4.2BSD or System V) is essentially defined by the System Interface layer, dealing with the direct handling of system calls. Inbetween these two layers is a System Services layer, which defines and provides the large scale facilities to support the system calls. The structure of these facilities is to some extent what distinguishes Unix from other operating systems.

From this perspective, if one wished to simultaneously support two Unix versions, the major changes must be made to the System Interface layer. Some modifications may be required of the System Services layer; these changes tend not to be drastic overhauls but additions of new features. (System V's semaphores and messages are examples of interface changes which impact this layer.) Virtually no changes must be made to the Machine Interface layer.

This characterizes the approach we've taken. Because the Unix version impacts largely the outer layer, we can select the best Unix implementation in terms of speed and capabilities for the internal layers and still support 4.2BSD and System V. For the inner layers, we've chosen a fairly strict implementation of 4.2BSD for the following reasons:

- It is the only Unix version which supports virtual memory and demand paging on large machines. We've modified the 4.2BSD approach somewhat to take advantage of hardware features available on our architecture, and to provide larger virtual spaces for processes.
- It is the only Unix which has really addressed the Unix I/O bottlenecks. For running on a large supermini, we felt the fast file system implementation was a necessity.
- Its network facilities are considerably more extensive than other versions. We believe that distributed processing will be predominant in the future, and this network support provides the framework.
- There are a variety of other smaller features that are nice to have -- sockets, flexible length file names, more powerful signal mechanisms, etc.

In modifying the System Interface and System Services layers, there are several fairly difficult conflicts to resolve, and a host of smaller changes which are straightforward to implement. Included in the former class are:

- Differences in terminal driver design and how character I/O is controlled.
- Signal handling
- System V interprocess communication
- FIFO's (named pipes)
- Incompatibilities due to flexible file names in 4.2BSD vs fixed file names in System V.

These will be described in some detail in the sections which follow. The other changes will be discussed only in terms of the general design heuristics we followed. These included:

- System call translation: It is sometimes possible to translate slight conflicts in formats or information returned from a system call from one UNIX version to the other. Because we wanted neither 4.2BSD nor System V to suffer performance penalties from the coexistence of the other, this "layered" approach is taken only when no significant loss in efficiency was involved. This layering can occur at either the system call stub within libc or within the System Interface layer. Because of the 4.2BSD basis of our system, all of the translations were from System V to 4.2BSD. Examples of this heuristic are *time, ulimit, dup,* etc.
- Separate 4.2BSD and System V system call entry points: This is a convenient method of providing to the kernel the knowledge of which version is making the request. This can be passed when necessary to the System Services layer to allow differential handling. Examples of this approach were setpgrp, kill, signal (discussed further below).
- Superset structures: System V and 4.2BSD differ slightly on several system header files (e.g., acct.h, sgtty.h, etc.). In most of these cases, the versions have a common set of structure members with the same names, with each having a few unique members for other functions. By using a system header which is a superset of the common and unique members from each version, utilities and applications programs running under either 4.2BSD or System V can run transparently to the existence of the other system. Typically, the amount of time required by the kernel to fill the superset structure vs the original structure is insignificant.
- Addition of new independent modules: Some of the features of System V are unique enough that it is easiest to incorporate new modules to the System Interface and System Services layers. The major examples of this are the System V semaphores, interprocess message facilities, and shared memory features. It would have been possible to layer these on top of the 4.2BSD IPC mechanisms, but this was both inefficient and leaves one open to small unanticipated incompatibilities. The amount of additional code to implement these features directly is relatively small, and the impact of interactions on the rest of the kernel is remarkably little.
- Special case algorithms: Some conflicts are quite deep. These require designing special mechanisms to handle each in a manner which maintains compatibility and doesn't sacrifice performance. These are described in the sections below.

To summarize the effect on the kernel of adding System V compatibility, a mirror set of System V system call entry points were added, and the size of the kernel increased by approximately 9%.

5. DIFFERENCES IN DIRECTORY NAMES (STRUCTURE):

The significant difference here from a systems interface viewpoint is that 4.2BSD uses a variable length buffer to hold the character names of inodes, while System V expects the directory name buffer to be of fixed length (DIRSIZ).

We believe that 4.2BSD's flexname approach to directories is potentially of great advantage, especially when networked applications become more common, and have kept that full capability in OSx. The problem is then to provide a directory interface to System V utilities and applications compatible with what they expect, while still retaining a flexname directory structure underneath.

The System V directory structure is as follows:

```
struct  direct
{
        ino_t           d_ino;/* inode number */
        char            d_name[DIRSIZ];
};
```

4.2BSD uses a variable length structure with an entry, d_namlen, which essentially defines the length of any given directory record:

```
struct  direct
{
        u_long          d_ino;     /* inode number */
        u_short         d_reclen;     /* length of
                             * this record */
        u_short         d_namlen;     /* length of this
                             * d_name string */
        char            d_name[MAXNAMLEN + 1]; /* directory string */
};
```

To solve the conflict between the systems, we take advantage of the fact that we know the program environment that the current process is living in. When a process issues a read system call, a flag is set in the process structure indicating that this is a System V read. If the read ends up calling *rwip* (as is the case when reading a directory), this flag is checked and, if this is a System V read, the variable length directory structure read from the disk is converted to a fixed length record which is returned to the user in the following structure:

```
struct  direct
{
        ino_t           d_ino;
        u_short         d_reclen;/* (unused) */
        u_short         d_namlen;/* d_name string length */
        char            d_name[DIRSIZ];
};
```

For applications programs, this is compatible with the original System V structure. We have increased DIRSIZ to be compatible with MAXNAMLEN in 4.2BSD so that a System V user will never see a 4.2BSD directory name truncated. This entails no performance penalty in copying the string to user space since the valid characters in a name are terminated by a NULL.

The net effect on System V users is that they can think in fixed length directory records while living on top of a variable length directory file system. (Programs which use the constant 14 rather than DIRSIZ will need to be modified slightly, but these programs should be caught anyway.)

6. FIFO's (NAMED PIPES)

Named pipes are a feature unique to System V whereby two processes can communicate by a pipe without knowledge of the file descriptor designation at the end of the pipe. 4.2BSD has implemented pipes in terms of the more general socket IPC mechanism, but did not implement this particular form of pipes.

We've implemented named pipes by adding a new inode type, IFIFO, (a la System V) and using the 4.2BSD socket mechanism to effect the actual piping, in much the same way that unnamed pipes are currently implemented in 4.2BSD. This entails making changes in the code for opening, closing, reading, and writing from inodes (and utilizing the socket functions *socreate, sosend, soreceive,* and *soclose.)*

A side effect of this implementation is that 4.2BSD users can take advantage of named pipes if they so choose. In the interest of a clean 4.2BSD interface, this feature is unadvertised in the 4.2BSD world, however.

7. SIGNAL HANDLING

There are a few major and a variety of minor differences between how 4.2BSD and System V do signal processing. In terms of signal types 4.2BSD is nearly a superset of System V, and again provided the starting point for our signal code. The most significant differences are:

- In 4.2BSD, signal handlers are set and cleared through a library subroutine, which translates requests into sigvec calls and saves signal masks. In System V, signal handlers are set and cleared directly through a signal system call.
- In System V, the function value of SIGCHILD can be set to SIG_IGN. If a wait is then executed, it will block until all children of the calling process have died.
- When a signal is caught in 4.2BSD, further signals are held or blocked. Neither option is available in System V.

We've modified 4.2BSD to provide an additional *signal* system call. Upon entry into the kernel, a flag is set in the process structure for that process indicating that the process wants System V- styled signal handling. (Note that whether a process invokes System V or 4.2BSD signal handling is independent of which universe the user is logged into.) For example, *wait* checks this flag before deciding how to handle SIGCLD.

8. SYSTEM V INTERPROCESS COMMUNICATION

In the area of IPC, System V introduced three new mechanisms, semaphores, messages, and shared memory, along with the associated system calls. For purposes of efficiency, and to avoid subtle unanticipated incompatibilities, we incorporated these features directly into the kernel, rather than layering them on top of sockets. In the case of semaphores and messages, the code ported almost without change from System V. The shared memory feature required porting to our virtual memory system (we were able to take advantage of hardware support within the 90x for shared memory pages). The first two cases require virtually no changes to the rest of the kernel. Shared memory is a new type of virtual segment, and, as such, must be specially handled by the pager and swapper. However, if such features are not used (as in the case of a strict 4.2BSD user), there is no impact at all on performance.

9. TERMINAL DRIVERS AND IOCTL

Providing dual 4.2BSD and System V compatibility for the operation and control of terminal I/O was perhaps the single most difficult compatibility task in implementing OSx. Although the interface through the *ioctl* system call is essentially identical in the two systems, the degrees of control provided and the underlying implementations are radically different. In this case, neither system is a superset of the other -- each provides a subset of unique features as well as common ones.

Although we make no claims about the ability of a user to freely switch between System V and 4.2BSD universes without subtle problems arising, we anticipate that users will attempt to take

advantage of both universes. Therefore, in addition to supporting both System V and 4.2BSD views of terminal I/O, we had a secondary goal of allowing users to switch back and forth between universes without leaving their terminal in unexpected states. The potential difficulties here can be illustrated by the following example.

We sign on into a System V universe, and do an *stty* to set the IOCTLs on our terminal to some value. We then decide to enter the 4.2BSD universe and execute some program there which requires modification of the terminal state (e.g., from cooked to raw mode, as in the case of an EMACS-like editor). This program does a *gtty* to save the state of the terminal, then does a *stty* to set the terminal in the state needed for the program. This brings up the first problem:

- We are now in a 4.2BSD universe, which does not know about part of the state vector known to the System V universe. Thus, when the *stty* system call above is made, only the 4.2BSD portion of the state vector will be meaningful.

  This can be solved by letting the kernel choose default settings for the rest of the state vector such that the 4.2BSD terminal will behave in a rational manner. During the execution of the 4.2BSD program, it may issue additional *sttys* to modify the terminal state. In each case, the non-deterministic part of the state vector can be treated in the same manner, using reasonable default settings. (Note that the kernel must take note of the universe of the process issuing the *stty* to know whether part of the state vector is non-deterministic.)

  At some point we now exit this 4.2BSD program and return to our System V environment. Before exiting, the program issues a final *stty* to restore the terminal to the state it was in when entering the program. This introduces a second, more difficult problem:

- The final *stty*, by which the 4.2BSD program intends to restore the System V state of the terminal, differs from all other *sttys* issued during the life of the program in that the System V portion of the state vector is now meaningful. Yet the kernel has no obvious way of detecting that this state vector is different. If it follows the solution to the first problem, it will choose default settings for the System V-unique portion, destroying that part of the System V environment that the user is returning to.

Our solution to this particular problem is described as part of what follows.

Although it could certainly be argued that parts of the 4.2BSD terminal driver are less elegantly designed and written than the System V driver, it has been our experience that the 4.2BSD version provides significantly better performance, better hooks into hardware flow control, etc. We therefore decided to use the 4.2BSD driver as a basis. The maze of conflicting and partially agreeing IOCTL parameters were redefined to be non-conflicting (while not changing any of the symbolic names in either universe). This redefinition was done in such a fashion that the terminal driver can detect from the parameter whether the calling process was 4.2BSD or System V. The terminal state vector within the sgtty structure was modified to include a superset of the states known to the 4.2BSD and System V universes. Code was added to handle the System V-unique IOCTLs.

Finally, to handle the universe-switching dilemma described above, we've designed a means by which the kernel can detect which *stty* system calls should be interpreted as transitions between universes as opposed to settings within a single universe: A word is added to the sgtty structure which is transparent to user programs. In it, the kernel puts a 32-bit identifier each time an instance of that structure is passed as part of a *gtty* system call. This tag identifies the sgtty structure as being from a particular universe. On a *stty*, the kernel checks this tag. If the identification is not from the universe the process is currently running in, then the kernel interprets the *stty* as a transition of the terminal state to the opposite universe (with the sgtty structure being passed being the one saved earlier), and interprets the state vector appropriately, introducing partial default settings if returning to 4.2BSD and interpreting the full state vector if returning to System V.

It is of course possible for a malicious user program to defeat this mechanism (e.g., by writing into the 32-bit identifier). However, the only effect of this would be to leave the terminal in a potentially strange state (no worse than what can sometimes happen when a raw mode program is aborted). However, the intention is not to bar malicious users, but to allow users to move smoothly

between universes during a login session, and not have to worry about the potential complications of the conflicting terminal conventions.

The bottom line is that programs that are based upon either 4.2BSD or System V terminal I/O conventions should not have to be modified to run under OSx, and that the user can execute a combination of such programs within a single session.

10. CONCLUSIONS

In the future we will continue to incorporate the latest developments from Western Electric and Berkeley into OSx. In addition, we will be concentrating on evolving OSx in three major directions:

- Developing a fully networked distributed operating environment.
- Incorporating more powerful and efficient virtual memory such as mapped files, general virtual process segments, and copy-on-write forks.
- Designing operating system support for multiple closely coupled CPU's.

In summary, we believe that it is possible to implement a Unix operating system which has all of the performance advantages of 4.2BSD underneath and provides full compatibility with both 4.2BSD and System V at the interface level, and which can be efficiently updated to reflect new releases from Western Electric or Berkeley. OSx should not be viewed as yet another flavor of Unix but an attempt to provide a transition towards a single Unix operating system for large machines.

A Layered Implementation of the HP-UX Kernel on the HP9000 Series 500 Computer

Jeff Lindberg

Fort Collins Systems Division
Hewlett-Packard
3400 E. Harmony Rd.
Fort Collins, CO 80525
USA

(jbl@hpfcla.UUCP)

1. OVERVIEW OF HP-UX

An implementation of the UNIX operating system kernel, called HP-UX, has been layered on top of an existing operating system kernel for the HP9000 Series 500 computer. The mapping of UNIX functional requirements onto the capabilities of the underlying OS are presented in this article.

The HP-UX operating system is compatible with Bell Laboratories' System III UNIX, and supports most of the standard UNIX commands and libraries. A number of extensions are available, including:

- FORTRAN 77
- HP Pascal
- C
- HP's AGP 3-dimensional and DGL 2-dimensional graphics subroutines
- Ethernet compatible 10 Mbit local area network
- The 'vi' visual editor
- Virtual memory
- Shared memory
- HP's IMAGE data base management system
- Support for multiple symmetric CPUs

Another HP9000 family member, the Series 200, was recently introduced. All references in this article to the 'HP9000' refer to the Series 500.

2. SUN OPERATING SYSTEM KERNEL

When the HP9000 project was begun several years ago, the operating system designers took a different approach than that used on HP's previous desktop computers. Even though the first HP9000 system was to be an extension of the BASIC language system of the 9845 desktop computer, an objective of the operating system design was to allow other languages in later versions of the product. The system software was designed in a modular, layered fashion. A central operating system kernel provides a high level interface to the hardware and machine architecture, while other subsystems provide more specific functions layered on top of this kernel. This operating system kernel, called SUN, is described in detail in another article in this issue.

SUN is implemented mainly in Modcal, an enhanced version of Pascal. Modcal supports information hiding via modules, an error recovery mechanism, and systems programming extensions such as absolute addressing. A small part of SUN is implemented in assembly language.

The SUN kernel itself is not directly visible to the user; instead it relies on upper level subsystems such as BASIC or HP-UX to provide a user interface.

2.1. Major Components

The major pieces of the SUN OS kernel are the following:

- Memory management
- Process management
- File system
- Drivers
- I/O Primitives
- Real time clock
- Interprocess messages

An unusual feature of the file and I/O system is the ability to add new directory format structures, device drivers and interface drivers. These modules can be added without affecting the existing SUN kernel code.

2.2. Missing Components

Some key pieces are missing from SUN by design, notably the human interface and program loader. The BASIC system provides its own human interface code which uses the integrated CRT and keyboard of the Series 500 model 20. HP-UX provides a terminal-style human interface to communicate with the user through the integrated CRT/keyboard as well as through normal terminals. HP-UX and BASIC also provide their own unique program loading facilities.

3. Series 500 HP-UX KERNEL STRATEGY

The basic strategy of the Series 500 HP-UX implementation is to layer the HP-UX kernel definition on top of the SUN operating system kernel. The exact System III UNIX semantics and syntax are kept, but the HP-UX intrinsics are implemented using SUN kernel support instead of porting Bell Labs' kernel implementation to the Series 500.

A layer of code called the 'HP-UX layer' resides just above (and in some cases beside) the SUN kernel, as does the BASIC subsystem. The HP-UX layer performs any necessary transformations between UNIX formats and the corresponding SUN formats, e.g. real time clock format. It calls procedures in SUN whenever appropriate, but still has full access to the hardware and architecture when needed. The HP-UX layer maintains a number of higher level data structures which manage HP-UX user processes and user resources.

This layering strategy has a significant impact on the implementation details of the HP-UX layer. For example, Modcal is used instead of C as the implementation language. However, user level code written for System III UNIX will run on HP-UX, unless it depends on certain internal implementation details such as the directory format structure or invisible internal system data structures.

3.1. Benefits

The advantages of this approach for HP-UX on the Series 500 come in two main categories: leverage and opportunities for contribution.

A large portion of hardware dependent code was already written for the Series 500 and its peripherals. By using the SUN kernel, the re-implementation of this functionality was avoided in HP-UX. The modules 'stolen' included device and interface drivers (especially significant because of the complexity of HP-IB and the new HP CS80 discs), low level memory management, power-up code, process scheduler, architecturally dependent utility routines, and other machine dependent code.

SUN has a number of features which are not present in native UNIX; these features provide opportunities for HP-UX to make a contribution above and beyond other UNIX implementations. These include real-time performance in the area of interrupt response time and process switching, support for multiple CPUs, reliability in the face of system errors, support for variable-size

independently managed dynamic memory segments, semaphores, and low level device I/O capability (e.g. GPIO, HP-IB). Also, the IMAGE data base management system was already implemented on top of SUN for the BASIC system. This code has been ported to the HP-UX environment (for release 03.00) to provide this important HP standard data base capability.

3.2. Risks

Since the UNIX semantics were being reimplemented in Modcal code on top of SUN, there was a significant risk of incompatibility with System III UNIX. Thus an extensive validation effort was required to ensure compatibility, and to document known incompatibilities due to implementation details. The validation effort and the actual experience in porting UNIX commands and libraries to the HP-UX system proved the excellent UNIX compatibility of the HP-UX kernel implementation.

Another concern was performance of a layered implementation; the risk was that conversion between SUN format and HP-UX format would increase OS overhead. The experience actually observed after the product was completed was that the HP-UX layer itself is responsible for approximately 10 percent of the CPU time used by the kernel; nearly all of that time is spent doing useful work such as loading programs. This means that the SUN functionality is a fairly good match to the HP-UX requirements, since little time is being wasted on conversion between SUN and HP-UX formats.

Also, since SUN was not originally designed with UNIX in mind, the areas which had been tuned for performance were not necessarily those which would make the greatest contribution to performance in a typical UNIX system.

4. MATCH BETWEEN SUN AND HP-UX

This section describes the areas of SUN that were changed or augmented in order to support the requirements of HP-UX. Only those areas which are important to mapping the UNIX semantics onto the original SUN kernel are described in depth. Little details are given in HP-UX extension areas, and the function of the HP-UX layer itself is generally straightforward so that details are unnecessary.

Some of the additions mentioned are actually maintained separately by the HP-UX layer development engineers, but the code is considered to be at the SUN level. For example, some of the *fork* code is really at the SUN level because it deals directly with segment tables and other low level data structures.

4.1. File System

There was already a good match between SUN and HP-UX in the hierarchical directory structure of the file system. This existing directory format was modified to fit HP-UX semantics rather than implement the native UNIX disc format in Modcal. The fundamental operations such as read, write, open, close, etc. were already supported in a satisfactory manner in SUN; no significant changes were necessary.

But the file system is the area which required the largest changes in SUN. One of the biggest additions was the support of *device files*, special files which map devices such as printers or terminals into the same name space as regular files. The SUN file system expected device and file accesses to be requested separately. Special checks had to be made for special file types; the new device file code performs operations for device files equivalent to those originally performed for regular files.

Another large change was support for mounting disc volumes onto a currently on-line directory, so that all accessible files and directories are part of a single directory hierarchy. Again, special code was added to check each directory access; if the directory has another volume mounted on it, the access is redirected to the root directory of the mounted volume.

The third area of major change was file access protection semantics. The UNIX read/write/execute and user/group/other mechanisms used to control access to files were not

originally in the SUN file system protection scheme. This could have been added, along with the native UNIX disc format structure, to a separate directory format module, since SUN supports multiple directory format structures. However, the characteristics of the existing format were so close to those desired that the SUN format and protection scheme was adapted to the HP-UX requirements.

Changes were made in the SUN file system to support pipes and FIFO files. In the first pass implementation of HP-UX, pipes were implemented in the HP-UX layer. However, they have been moved inside the SUN file system for performance reasons.

A number of minor HP-UX file system operations had to be added to SUN. These include changing the owner of a file, reading or changing file access modes, and duplicating an open file descriptor.

Some operations are performed in the HP-UX layer. These include parsing multi-level path names, managing the user's open files table, and enforcing file size limits on extending files.

4.2. I/O

In the area of device I/O, the existing SUN I/O system was a very good match for the needs of UNIX. Virtually no changes were made to the I/O primitives which provide the interface to the backplane and I/O processor, the bus bandwidth management code, the drivers for interface cards or the disc and tape device drivers.

The major changes came in the internal and external terminal support. The external terminal driver was based on the existing serial interface driver, but added UNIX TTY semantics such as type-ahead, line buffering, mapping carriage return/line feed to newline, and sending the *interrupt* and *quit* signals.

The integrated keyboard and CRT device control code was based on the work done for the BASIC system's human interface. But the functional operation of the integrated 'terminal' had to be completely redone to be compatible with HP terminals.

4.3. Memory Management

Because of the simple memory model of HP-UX, the memory allocation intrinsics are easily supported on most operating systems, including the SUN kernel. The major changes in the SUN memory management system were due to the addition of virtual memory capability, which is an extension rather than a semantic requirement of UNIX.

The HP-UX layer has the responsibility of keeping track of the user's memory usage and deallocating this memory when a process or program terminates.

4.4. Program Loading

No explicit function for loading and executing programs is present in SUN, but the underlying support needed is there. The file system is used (with minor changes) to find and read the program file, and the memory management system provides the mechanism for allocation of code and data segments. No major changes were required in the SUN kernel to support program loading.

The HP-UX layer manages shared code segments, which allow multiple processes to share a single copy of the code. The HP-UX layer also handles relocation of code and data segments at load time, and meeting the segment attribute requirements requested by the object file format.

4.5. Process Management

The HP-UX process management intrinsics are supported fairly well by the SUN kernel, but two areas required a significant effort: *fork* and *signals*.

4.6. Fork Implementation

The *fork* system call creates a new process in the exact image of the calling process. It returns to both the parent and child processes just after the *fork* call, at the point where the function return value distinguishes the child from the parent. Creating an exact copy of a process is not a typical operation supported by normal operating systems, including the SUN kernel.

At the SUN level, code was added to support the 'cloning' of a process. This code runs primarily on a separate system process for two reasons: 1) the parent's stack segment must be quiescent at the time of the copy, since it is very difficult to get an accurate picture of a moving object; and 2) the system process has the required addressability to the new child's memory. The parent process cannot gain the required addressability, since its stack resides in its own private address space, and only one private address space can be in effect at one time. The system process stack is in the shared address space, and so is still valid when changing the private address space.

The cloning operation running on the system process calls lower level SUN procedures to allocate memory for the child process and initialize SUN modules for the new process. It is also responsible for duplicating the contents of the parent's segment table in the child's segment table and creating an exact image of all the parent's segments in the child's address space. Special SUN kernel support is necessary to clone the virtual memory segments.

There is also a special version of the process creation code which creates a child process using an existing stack, and causes the child to inherit attributes from the parent.

The HP-UX layer calls this SUN level code to clone a parent process, and then executes other code at the HP-UX level to initialize the new process. This includes allocating an HP-UX process control block, copying some fields from the parent's process control block, and initializing other unique fields such as process ID and parent process ID. It also increments use counts on shared objects such as shared code segments and open files.

Finally the HP-UX layer returns the appropriate function value to the parent (child's process ID) and to the child (zero).

4.7. Signal Implementation

The implementation of signals, primarily the sending and processing of signals, was a significant portion of the HP-UX layer development. SUN had no explicit support for sending asynchronous signals between processes, but did have most of the tools necessary to implement this feature.

One tool is the ability for subsystems to install trap handlers for most classes of traps possible on the Series 500. Signal processing is initiated by triggering an MI (Machine Instruction) trap in the target process, which causes the MI trap handler to be entered on the next machine instruction executed. The MI trap handler is responsible for processing the signal received, and taking the specified action. This can be calling a user-specified signal handler, terminating the process, or just ignoring the signal.

The process scheduler triggers an MI trap in the process about to be dispatched if a signal is pending. The assured periodic interruption of each CPU by the SUN timer interrupt ensures that even a process which is in an infinite loop in user code can receive a signal. The minor changes made to the SUN level code were: a new field in the SUN level task control block used to log the receipt of signals by a process; and the new code in the process scheduler which checks for pending signals.

If the target process is in a known blocked state, the process is forcibly unblocked. Subsequently the MI trap handler processes the signal.

The HP-UX layer code includes the MI trap handler which processes signals, as well as the code which sends signals to one or more processes. It also handles specification of the action to be taken upon receipt of a specific signal, and blocking until a signal is received.

4.8. Other Process Management

The process scheduler met the requirements of HP-UX in the original SUN implementation, but has been improved to allow dynamic process priority adjustment to reward interactive processes. SUN supports the creation of special system processes which can provide specific system services. These system processes communicate with user processes and each other via SUN's mailbox-style interprocess messages. Also, a sophisticated set of semaphore operations is provided for synchronization of all processes in the system. This is especially important in a multiple CPU system; merely disabling interrupts does not ensure exclusive access to a shared data structure, since other processes may be running simultaneously on other CPUs.

The following process management functional areas are implemented in the HP-UX layer: Higher level support of *fork*, such as allocation and initialization of a process control block for the new HP-UX process.

Higher level support of signals, including sending and receiving signals, and specifying action to be taken on receipt of a signal.

Management of user, process and group IDs.

Process termination, including deallocation of resources owned by the user process.

Wait for a signal or for termination of a child process.

Management of HP-UX process control blocks.

4.9. Other

The functional areas listed below were completely supported by the SUN kernel, except for those changes noted.

- Powerup.
- Multiple CPU support.
- Trap handling.
- Real time clock; the HP-UX layer performs the conversion between SUN time format and HP-UX time format.
- Alarm clock; the HP-UX layer creates a system process which wakes up each second to see if any alarm signals need to be sent.
- CPU times; a minor change was made to the timer interrupt service routine to increment the CPU time used by the current process.

4.10. Tools

The existence of system-software development tools for Modcal and the SUN kernel environment was a significant factor in bringing HP-UX up quickly on the Series 500. Tools available included the Modcal compiler, assembler, linker and other utility programs, as well as a powerful symbolic debugger for use in developing system software. These tools were used in HP-UX kernel development without change.

5. LIKELY PROBLEM AREAS FOR LAYERED UNIX IMPLEMENTATIONS

The areas likely to cause the greatest grief in layering UNIX on top of an existing operating system are listed below: Those marked with a '*' are likely problem areas even for a 'UNIX-like' implementation which provides only a subset of UNIX capabilities.

- Hierarchical directory structure
- Mounting disk volumes onto an on-line directory
- Multiple links (alternate names) to a file

- File access protection semantics
- Interprocess pipes
- Device files in the regular file name space
- UNIX terminal semantics
- Process creation via *fork*
- Signals

Most of the problems likely to be encountered in the above areas were described earlier in this article.

6. CODE SIZE

The Release 2.0 HP-UX kernel layer includes approximately 45 Kbytes of object code, not counting the 9020 integrated terminal emulator code (approximately 25 Kbytes). For comparison, the SUN kernel contains roughly 175 Kbytes of object code, not including any optional device or interface drivers. These numbers do not include HP-UX extensions to standard UNIX such as IMAGE data base management or networking.

An Intelligent Windowing Graphics Terminal for the UNIX System

Teletype Corporation

M.J. Kelly

AT&T Bell Laboratories
5555 Touhy,
Skokie IL 60077,
U.S.A.

F.A. Saloman *Thomas D. Rhodes* *William S. Goldberg*

AT&T Bell Laboratories
1100 Warrenville Road,
Naperville IL
U.S.A

ABSTRACT

An important feature of the UNIX System is per-user multiprogramming; that is, each user may control several concurrently executing processes. However, this feature breaks down at the user interface. Solutions such as job control, background processing and others do not address the problem of maintaining several dynamic display contexts.

The **TELETYPE 5620 DMD** is a high-resolution bitmapped display terminal. The 5620 screen may be divided into a number of rectangular windows, called *layers*, each of which appears to the UNIX System host as a separate terminal. A multiplexing communications protocol allows each layer a private full-duplex communications channel to the host. Layers may arbitrarily overlap, yet all are concurrently updated. Graphics operations in a layer are automatically performed in the visible portion and in all affected obscured portions of the layer. The result is that changes are immediately visible and the obscured portions are always up-to-date.

This paper briefly describes the layer concept, discusses the implementation of 5620 software on UNIX System V and shows some examples of the usefulness of the 5620.

1. Introduction

An important feature of the UNIX System is per-user multiprogramming; that is, each user may control several concurrently executing processes. Unfortunately, this feature breaks down at the user interface. UNIX systems rely on 'dumb' or semi-smart terminals as the primary user interface, and these are not able to maintain several concurrent interfaces. The solution found in 4.2BSD (job control) still does not adequately solve the problem of maintaining display contexts for concurrently executing processes.

The practical result of this limitation is that few people routinely run several concurrent processes. What is needed is a way of 'cloning' the terminal as a new process is started on the host. In this way, each host process would have its own communications line, display and keyboard.

The TELETYPE 5620 DMD provides exactly this ability to clone a (virtual) terminal on demand. The 5620 is a high-resolution bitmapped display terminal based on a 32-bit microprocessor (WE 32001), 256K or 1M bytes of dual-ported RAM (of which any contiguous 100K bytes is the display memory) and 64K-256K bytes of EPROM.

The 5620 screen is divided into a number of rectangular windows, called *layers*, each of which

appears to the UNIX System host as a separate terminal. A multiplexing communications protocol allows each layer a private full-duplex communications channel to the host. Layers may be created with a mouse, or by host software with the *ioctl*(2) system call.

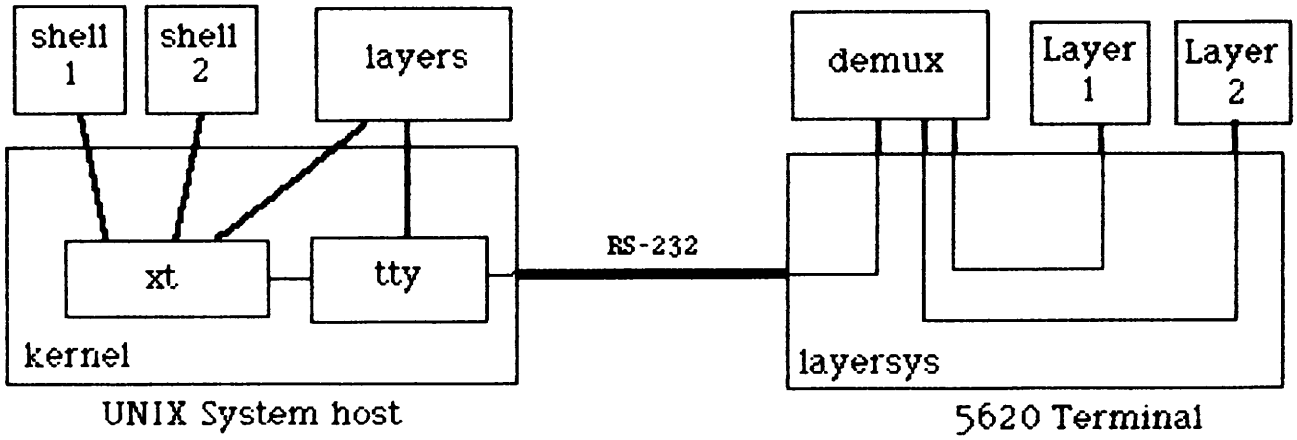


Figure 1. Management of Layers.

Layer A obscures Layer B. The rectangle (r) is stored on-screen for the top layer, A, and off-screen for layer B. Flipping visibility (so that B is on top) is a simple matter of swapping the rectangles and linking the off-screen rectangle to A, rather than B. (Figure from [1]).

2. Layers

Layers [1] are rectangular windows which may arbitrarily overlap and are all concurrently updated. The 5620 implementation of layers divides layers into visible and obscured rectangles of bits, and maintains the obscured rectangles of a layer in off-screen memory (Figure 1). The graphics primitives (*line*, *circle*, *rectangle fill*, etc.) are performed in all affected portions of the layer, visible and obscured. The result is that changes are immediately visible and the obscured portions are always up-to-date.

Several operations can be performed within layers. The most important is *bitblt* (bit block transfer). *bitblt(sb, r, db, p, f)* copies a rectangle of bits *r* in the bitmap *sb* to a congruent rectangle with origin *p* in the destination bitmap *db*. The nature of the copy is specified by the function code *f*; for example, *f* = XOR produces in the destination rectangle the exclusive OR of corresponding bits in the source rectangle and the destination rectangle before the *bitblt*. Theoretically, if not in actual implementation for efficiency reasons, all other graphics operations reduce to a sequence of *bitblt* operations. Four layers are present. At the top is a layer running *jim*, with two textframes

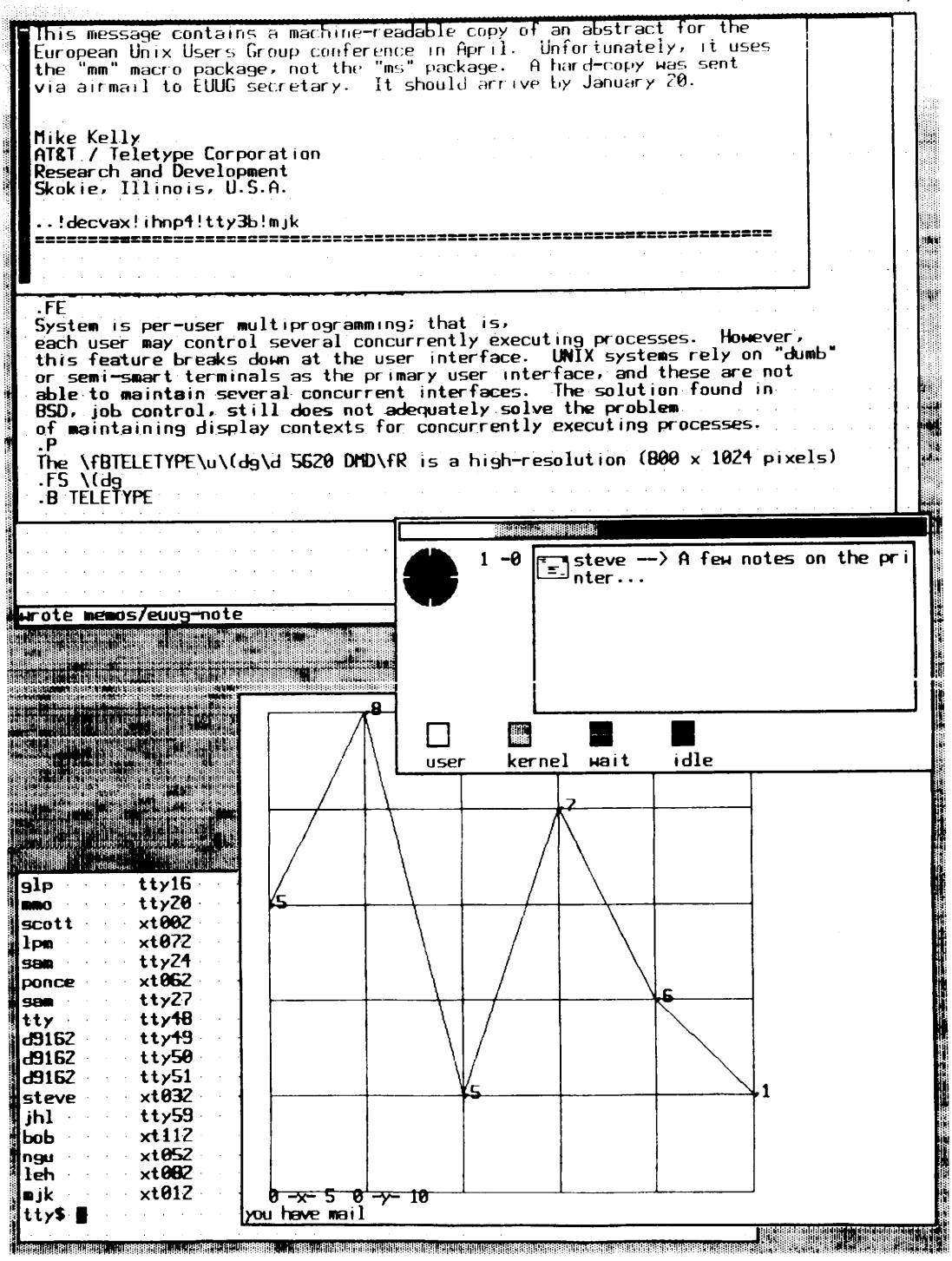


Figure 2. A 5620 Screen.

open. The active textframe is marked with the scroll bar (the inverted video vertical bar along the left edge). At the bottom left is an obscured terminal layer, running *who*. Overlaying that layer is one running the 4014 emulator showing sample output from *tplot*. Finally, in the middle of the screen is a layer running *sysmon*, a system performance monitor. The bar graph along the top, which is periodically updated from the host, gives an idea of the time spent in user, kernal, wait and idle states, respectively. The clock shows the current time, and the rectangle to the right is updated when mail arrives (with the sender's name and the subject line of the received message.)

3. The Layers Environment

The 5620 screen is shared by up to six layers; unused screen areas are shaded with a distinct background texture. (See Figure 2) Layers can be used in two ways: as a standard 'smart terminal' or as an intelligent application-customized terminal. Existing UNIX System programs expect a standard terminal, and these will run unmodified in 'smart terminal' layers. Software written specifically for the 5620 usually automatically downloads a program to customize the layer to the application. The downloaded *layer processes*, typically written in *C*, are managed in the terminal by a small non-preemptive scheduler, called *layersys*. When a layer is created, it starts as a 'smart terminal' layer, controlled by a layer process called the default terminal program.

layersys provides the user with a command menu that pops up on the screen in response to a button push on the mouse. To select a command, the user points at an item in the menu with the mouse and releases the button. Some commands require two stages: first the command is selected, then the user completes the command with another mouse operation. For example, 'New' is a command to create a new layer. Once 'New' is selected from the menu, the mouse cursor changes to a 'box cursor', which visually prompts the user to sweep a rectangle with the mouse. The rectangle swept out on the screen becomes the display area for the new layer. Similarly, 'Reshape' allows the size and position of a layer to be changed, while 'Move' drags a layer intact to a new screen position. All of these commands are also accessible from the host through an *ioctl*(2) call.

4. Layers and the UNIX System

The layers model -- multiple asynchronous processes sharing resources -- corresponds well to the UNIX System environment. Since each of the layers appears to the UNIX System as a separate terminal, each may have its own host processes running. (See Figure 3) In fact, all that is lacking in standard UNIX System V is a multiplexing line protocol, which is needed for several unrelated programs to share a single physical terminal connection. The multiplexing protocol used by the 5620 is implemented with a new UNIX System driver, *xt*. *xt* supports a sequenced, error-checked packet-multiplexed protocol over standard asynchronous lines. It acts like a standard **tty** driver to most programs. A few *ioctl*(2) functions have been added for use by programs which are written specifically for the 5620. Other programs (e.g. *ls*(1), *ed*(1)) neither know nor care that they are talking to anything but a standard video terminal. The 5620 processes (only two are shown: Layer 1 and Layer 2) are one-to-one with process groups (denoted here by shells) on the host. *layers* is connected to a control channel, and it receives messages sent to unallocated channels. In the terminal, *demux* is a process handling the packet discipline. (Figure modified from [2].)

Special files for *xt* come in groups of eight. Each group supports one physical link; each special file corresponds to a single channel on a link. These files are named '/dev/xt*ll*c', where '*ll*' is the link index and '*c*' is the channel number (0-7). Channel 0 is a control channel, 1 is reserved for future use, and 2-7 are channels to the six layer processes. A user-level program, *layers*, establishes the host side of the protocol and downloads a small piece of *layersys*.

To illustrate the way the layers environment works, we'll describe the 'New' command in detail. Once the user has drawn the rectangle for the layer (this may overlap any layers already on the screen), *layersys* allocates a channel for the new layer. It sends a 'NEW' control message to the host on the new channel. It starts a default terminal program for the layer, which mimics a standard video terminal (i.e. it sends characters typed on the keyboard to the host, and displays characters sent from the host). Meanwhile, the 'NEW' message has been received on the host by the *xt* driver and passed along to the layers program, since it is a control message. layers *forks*, does a *setpgrp*, *close*s all its fd's and *open*s the new channel as standard input, *dup*s the fd for standard output and error, and *exec*s a shell. The shell starts running, sends a prompt out its standard output, which passes through the *xt* driver, is packetized and received by *layersys*. *layersys* acknowledges the packet to the host, and places its contents in the process input queue for the new layer. The default terminal program has blocked on host input, so it wakes up, reads the prompt characters from its input queue and draws them in the layer window. The user now sees a Shell prompt. This entire sequence takes less than two seconds.

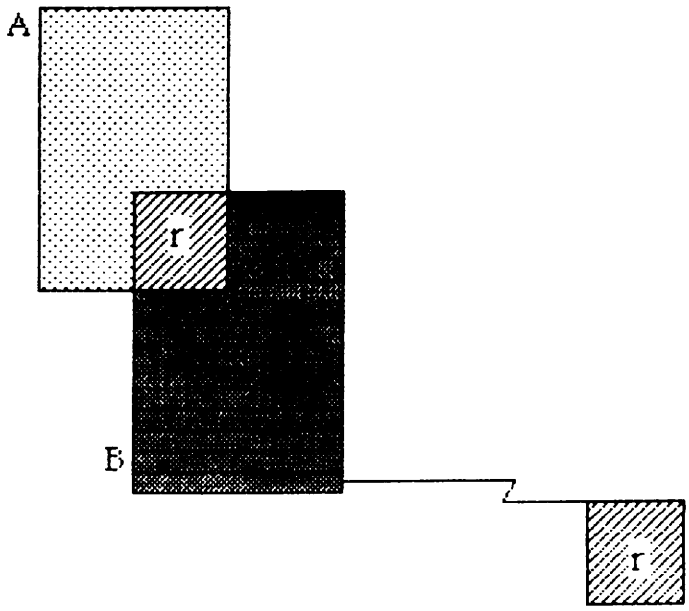


Figure 3. 5620 and UNIX System V.

At any point, the user may replace the default terminal program running for a layer with a different program. This is accomplished by invoking a host program (*32ld*) to download a WE-32001 executable into the terminal. An interesting point here is that there is nothing special about the default terminal program. It can be replaced by an emulator for some other terminal or a program specifically designed for some application. In fact, it is even possible to change the program *layersys* starts by default for a new layer.

5. Software

A number of applications have been developed for the 5620, including a mouse-based text editor, a Tektronix 4014 emulator, a remote file access facility for layer processes, a *troff* previewer and a picture editor. There is a UNIX System software generation system for the terminal, a layer process debugger, and several libraries of terminal functions (including an implementation of the standard *libc* for the terminal). Representative of this software are the three programs described below.

- A text editor (*jim*) which edits multiple files in 'textframes.' A textframe appears on the screen as a rectangle within jim's layer. Several textframes may be visible at once and, like layers, textframes may overlap. The mouse is used to select commands from menus and to select text for 'cut' and 'paste' editing. *jim* is a step beyond conventional screen editors. It uses the mouse as the primary editing device, replacing both cursor keys and hieroglyphic commands.
- A debugger for terminal programs (*dmdebug*) which runs in one layer while the program being debugged runs in another. The mouse is used as a replacement for command entry from the keyboard; menus contain commands and arguments. For example, the argument to the 'breakpoint' command (a name of a function in the program) is not typed, but selected from a menu of function names presented by the debugger.
- A picture editor which produces a description of the figure in *pic*,[3] a *troff* pre-processor. Figures are composed from a menu of shapes (circles, boxes, lines, curves) and multi-font text. The mouse is used to scale components and move them around until the desired picture is composed. The output may be included in any *troff* document.

Many more applications are under development.

6. Conclusion

The 5620 marks an important step in the evolution of the UNIX System: the development of a terminal customized to the capabilities of the operating system and its applications. To take full advantage of UNIX systems, a terminal able to maintain several display contexts simultaneously is needed. A bit-mapped display is needed for the growing set of graphics and high-quality text tools available. The 5620 provides this and more. Its programmability fits neatly into the principle of putting functionality in the right place by moving terminal processing out of the host and into the terminal. It provides workstation capabilities, but on existing UNIX hosts and over existing communications lines. The 5620 is a hardware tool for UNIX Systems.

References

1. Pike, R., "Graphics in Overlapping Bitmap Layers," *ACM Transactions on Graphics* **2**(2), pp. 135-160 (1983).
2. Pike, R., "The Blit: A Multiplexed Graphics Terminal," *AT&T Bell Laboratories* TM 83-11271-3. July, 1983.
3. Kernighan, B.W., "Pic: A Language for Typesetting Graphics," *Software Practice & Experience*, **12**, pp. 1-20 (January 1982).

The Norwich Renal Unit Programme

P. B. Pynsent and J. S. Pryor[‡].

School of Biological Sciences, University of East Anglia,
Norwich, NR4 7TJ.
[‡]Department of Renal Medicine, West Norwich Hospital,
Norwich, NR2 3TU, U.K..

1. INTRODUCTION

It is estimated that in Europe 120 people per million of the population suffer from chronic renal disease, of these 80% depend, for a variable period, on an artificial kidney machine for survival. The application of computers to manage renal-patient data bases is not new (Rorive *et al.*, 1980; Charlton, 1981; Stead *et al.*, 1983). We have developed a UNIX based computer system which not only provides access to a patient data base but also controls kidney machines during the haemodialysis of patients. The hardware comprises a PDP 11/23 with 128k words of memory, two RL01 disc drives and a Digidata 1740 magnetic tape unit. Information is input and displayed using VT125 terminals modified with touch sensitive screens (Interaction Systems Inc.).

2. THE DATABASE SOFTWARE

All user facilities are selected from a screen menu. For the hospital staff the first menu is displayed immediately after 'login' and a programme may then be selected by touching the screen or typing an associated number. After selection of a programme a new menu is displayed for the specified function, this menu includes the option to request a different programme. Movement from one programme to another is accomplished using a 'fork' and 'execl' in the normal manner. A programme called FIND is used to find a specific patient, this may be done using the full hospital number or alternatively by giving clue to the name, for example 'Sm' could be entered to find 'Smith', possible patients are displayed in sequence until the correct patient is found or the patient list is exhausted. Although FIND may be used *per se*, it is usually called automatically (using a 'fork') by other programmes when details of a specific patient are needed. Once a patient has been selected their name, number and date of birth are always displayed with a highlight attribute at the top of the screen, these details are also passed with 'execl' between different programmes. A HELP option may be selected from the main menu to give help on how to use the system.

A patient admission programme sets up a patient directory and creates the required files for their medical data. The patient is uniquely identified by their six digit hospital number. The ADMIT programme also asks for other details of the patient such as date of birth, address and next of kin. If the number given is already known to the computer then it is assumed that the data is to be changed and a screen editor is made available.

A programme called by touching TABULATE displays patient data in tabular form. Data such as blood presure, weight and blood biochemistry are displayed as a table on the VT125, this table may be smooth scrolled up (forward in time) or down using two touch areas on the screen or by use of the arrows on the keyboard. The actual data to be displayed is selected from a second menu. PLOT is similar to the tabulation programme only that in this case the data is displayed in graphical form and the plots may be scrolled across the screen through time.

A special screen editor is provided for adding and changing patient data. The patient data base is stored as files of random length records. Data are always accessed with respect to the date the information was obtained from the patient, for this reason the index associated with each data file links data by date. Thus an entry on a data file contains a pointer to the next and previous (with respect to time) entries in addition to the data.

Other programmes available to the user include the normal Unix text processing facilities with output being spooled to a high resolution matrix printer (Sanders Technology 12/7). A programme has been written specifically for the dietitians, giving access a data base of over 1000 different foods (Paul and Southgate, 1978; Wiles *et al.*, 1980). Foods are selected by one or two string keys. For

each food 38 items of data such as the energy, nutrient and electrolyte content are available to the user. The dietitian often needs to take patient diet histories, for this reason the programme supplies facilities for adding a sequence of foods of given weights. These food content totals may be displayed at any time, also printing of these totals in the form of a patient diet report is possible.

3. DIALYSIS CONTROL SOFTWARE

Removal of water from a patient with chronic renal failure is a basic therapeutic requirement. This procedure often leads to the patient vomiting, getting muscle cramps and low blood pressure. We have endeavoured to to minimising these unwanted effects associated with water removal by applying various programmes. The total amount and rate of water removal has been difficult to control until the recent development of microprocessor controlled kidney machines. The introduction of digital processing has meant that the interfacing of a kidney machine to a central computer is now possible.

We have developed a programme which enables the nephrologist to create and maintain a dialysis prescription for a particular patient patient. The programme plots the currently prescribed time-course of water removal and has screen functions which allow this to be changed. Linear and non-linear functions for the rate of water removal may be defined. The dialysate sodium concentration and duration of dialysis may also be changed, any changes are displayed graphically. When a final prescription is decided it is filed away for use when that patient is next dialysed.

Monitoring of patients haemodialysis session is at the nurses station on the unit. The monitoring programme displays a series of boxes, each representing a bed position. By touching a box more detailed information about a patient may be obtained or new information added. An initial touch of a box runs the programme to find a patient, subsequent touches move through a standard sequence, requesting patient weight, blood pressure and type of kidney machine to be used. If the patient is connected to a computer controllable machine and a model has been defined in the prescription for that patient, then the monitoring programme spawns a child process to control that patients dialysis. This child process has a two way communication (pipe) with the parent. The main monitoring programme updates the screen every thirty seconds adding any information (such as an alarm condition) it has received from the spawned processes. After haemodialysis of a patient is completed the programme spawns a process to plot a summary of the dialysis session on a plotter (Calcomp 81). The software allows for monitoring at a local nine bedded ward and a remote two bedded acute unit (via a leased PSTN line). In the future we will expand our computer facilities to encompass the monitoring of patients dialysing at home and in small minimally staffed remote units.

4. CONCLUSION

The objective of the Norwich Renal Unit project is to improve patient care using computer technology. First we have provided facilities for computer controlled kidney machines to optimise dialysis therapy to the individual patient. Secondly, we have provided an easy to use patient data base to aid the physician in his assessment of patients. The Unix operating system has proved an ideal environment satisfying both the multi-tasking and data processing requirements of our project.

References

1. Charlton, B. A. (1981). A nurse managed computerized medical record system in dialysis and transplantation. Proc. European Dialysis and Transplant Nurses Assoc., B9,B 132-135.
2. Paul, A. A. and Southgate, D. A. T. (1978). McCance and Widdowson's- The Composition of Foods. (London: Her Majesty's Stationary Office; Amsterdam and New York: Elsevier/North Holland).
3. Rorive, G., Gyselynck-Mambourg, A. M., Sabatier, J. and Gentinne, J. L. (1980). Computer-assisted medical care in a haemodialysis unit. Med. Inf. (London), B5,B 227-235.

4. Stead W. W., Garrett, L. E. and Hammond W. E. (1983). Practicing nephrology with a computerized medical record. Kidney Int. B24,B 446-454.

5. Wiles, S. J., Nettleton, P. A., Black, A. E. and Paul, A. A. (1980). The nutritional composition of some cooked dishes eaten in Britain: a supplementary food composition table. J. Hum. Nut., B34,B 189-223.

A Comparison of the UNIX and APSE Approaches to Software Tools

A. Burns and I.W. Morrison

Postgraduate School of Computer Science
University of Bradford

(alan@ubradcs.UUCP, iwm@ubradcs.UUCP)

ABSTRACT

This paper is concerned with the approaches to software tools employed by UNIX and the Ada Programming Support Environment. Within the comparison, recommendations are made which can be applied to the more generalised Integrated Project Support Environment.

1. INTRODUCTION

In 1973 the US Department of Defense undertook a detailed review of their expenditure on software. They found that $3 billion was being spent annually; by 1990 this figure was expected to rise to over $40 billion [1]. Significantly a large proportion of this outlay was taken up by maintenance. The use of over one and a half thousand computer languages and a great variety of operating systems was felt to contribute to this unacceptable cost. The result of this review was the funding of an ambitious project to obtain a new real-time programming language. This language, in 1979, became known as Ada; an ANSI standard for Ada was granted in 1983. Ada was designed as a single language to be used in all embedded systems design and implementation.

Another important aspect of the Ada project is the attempt to design and implement a standard support environment for the development and maintenance of Ada programs. The history of the development ideas behind the Ada programming support environment (APSE) is well documented in Elzer [2], Buxton [3] and Kramer [4] and the initial outline of the APSE is described by the Stoneman document [5].

2. WHAT IS AN ENVIRONMENT?

The term environment is generally accepted as meaning the aggregate of objects, conditions and influences that affect the existence or development of someone or something [6]. A software engineering environment consists of two important facilities; a methodology and the tools cycle [7]. It consists of a set of techniques to assist the developer(s) of a software system, supported by some (possibly automated) tools, along with an organisational structure to manage the process of software production [8]. Osterweil [9] discusses various situations which need to be supported in an environment:

(i) production - programmers working in the early stages of the design of a distributed real-time processing system need tools for creating, adjusting and verifying the system design etc.

(ii) management - where several programmers are working on differing projects, there is a need for resource allocation and in scheduling completion

(iii) verification - to develop a systematic approach that ensures the accuracy of the finished software, not only in being error-free but also conforming to the original specification

(iv) documentation - formal methods need to be adopted to allow comprehensive and clear documentation of the life cycle

(v) maintenance - large numbers of programmers working on code written by others which needs maintaining over a long life cycle. Tools are required to analyse code, maintain version control, test and verify changed code etc.

Such situations arise from varying degrees of complexity in software projects. McDermid and Piphen [10] summarise the criteria of an undemanding and of a demanding embedded project, and this is show in table 1.

Criteria	The Undemanding Project	The Demanding Project
size of application (lines of source code)	under 2,500	over 50,000
complexity of application system	low	high
total number of modules	under 50	over 1,000
during life of system life expectancy of system	under 10 years	30 years
reliability requirements	high	very high
configuration of host	centralised/ local network	highly distributed
configuration of target	single site and single processor	many sites and distributed hardware
size of development team	3 (distributed)	over 100

Table 1

One example of a large project is the Facilities Assignment and Control System (FACS) [11] where over 2,000,000 lines of C have been developed by over 100 people taking them 6 years.

When developing large software projects, several recurrent problems have been encountered. Bowles [12] summarises these as:

(i) the divergence of intended program behaviour and actual program behaviour

(ii) differences between project costs and estimations, frequently these were being exceeded

(iii) late delivery

(iv) logical errors, unreliable program execution - as program execution controls equipment or situations involving human life, the prevention of errors is of prime importance

(v) high maintenance costs in comparison with creation costs

(vi) duplication of effort on different machines or differing applications

These situations have lead to a software crisis, namely the need for:

(a) better languages

(b) better methodologies

(c) support environments

In a programming environment the tools developed to solve the above problems are integrated to provide continuous support over the entire software development cycle. Software tools themselves have for sometime played an important role in software production. Kernighan [13] describes tools as using the machine to solve general problems; with good tools being those that people re-use rather than re-invent. Many methodologies have been employed to aid in the design of software tools [14]. Indeed today in computing, there is realisation that the development environment plays a greater role than the set of individual software tools. Both UNIX and APSE are examples of development

environments, however they have differing approaches to the application and availability of software tools.

3. THE APSE

Buxton [15] in his rationale for Stoneman describes the requirements of the APSE. These requirements are made in order to meet the stringent characteristics of embedded computers in weapon systems. Real-time interaction, such as the monitoring and control of unique system components introduces timing dependencies; so there is a requirement for responding to external events (interrupts) within specified times. In such cases there is also the need for special interfaces. The computer used in the embedded system is designed more for the operational environment rather than for software development. This introduces the concept of host/target software development and of host/target machines. Source code developed for embedded computer systems is rarely classified, however there is a need for security of the data, as typically this can be of a sensitive nature. For example, the data used by one program might be the area to be covered by a plane on a specific mission. Considerations in this paper are however focused on those aspects of the APSE that are important for all areas in which Ada will be used, indeed the features discussed are arguably necessary in any integrated project support environment (IPSE).

The Ada language has been developed to provide both program and programmer portability. Hence a programmer moving to another environment will expect a consistent interface with portable tools. Furthermore, the environment must support the entire software life cycle. It must provide a coordinated set of tools which are applicable to all stages of program development. The interface between tools must be independent of the host machine. An APSE must also provide a uniform inter-tool interface, with individual tools communicating with each other, as well as access to the application programs, being made through a common database which acts as the information source and product repository for all tools. It should therefore facilitate the development and integration of new tools and the improvement, updating and replacement of tools. The Stoneman approach to portability is to create a three level model. This has the following features: a database (acting as the central repository for information associated with each project throughout the project life cycle); the user and system interface (which includes the control language which presents an interface to the user as well as system interfaces to the database and toolset); and the toolset (which includes tools for program development, maintenance and configuration control as supported by an APSE). The levels play specific roles (see Figure 1):

(i) KAPSE - Kernel APSE - provides a machine-independent portable interface which contains the database, communications and run-time support functions to enable execution of Ada programs

(ii) MAPSE - Minimal APSE - provides a minimal set of Ada written tools, supported by the KAPSE, which are both necessary and sufficient for the development and continuing support of Ada programs

(iii) APSE - provides the full support for a particular application or methodology

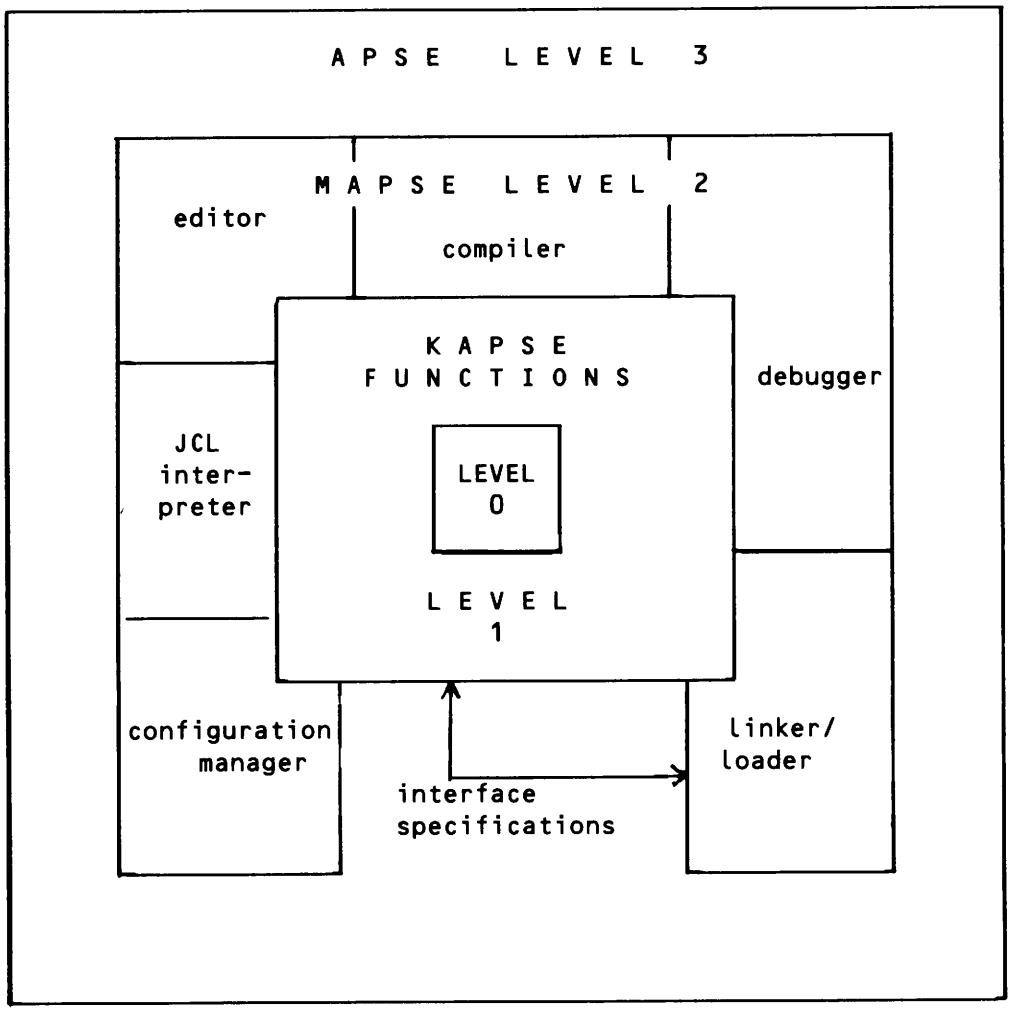


Figure 1

All software above the KAPSE is written in Ada, however the KAPSE itself can be implemented in any language thus creating a virtual machine. New tools can be added directly onto the KAPSE interface or be built on top of existing tools, and subsequently transported to other APSE's. However a tool developed in an APSE from existing APSE tools can prove difficult to transport without taking the set of related tools. Tool interfacing in the APSE is controlled via the communication to and from the database. The structure of the database facilitates the manipulation of attributed objects. An object is a separately identifiable collection of information. For example an object may be tagged to indicate that it is Ada source or DIANA [16] (intermediate representation); a tool such as a pretty printer is then defined to work on an object with DIANA attribute to produce one that has attribute Ada. Although a number of operating systems use name extensions on files to achieve this tool interfacing an APSE requires a secure system that will not necessarily allow the owner of objects to change or even, in some instances, delete them. All file access must be by appropriate tools. For example, we can consider the following scenario:

Object X *(attribute* "text" ...)
Tool Ada_Compiler (works on "text")
X ---> Y *(attribute* "DIANA" ...)
Tool Pretty_Printer (works on "DIANA")
Y ---> Z *(attribute* "ada" ...)

Z *must* contain syntactically correct Ada source.
Y *must* contain DIANA.

A restricted set of tools are allowed to act on Z and Y. The *relationship* between X, Y and Z will be retained by the database.

The database approach has a number of advantages over the traditional hierarchal file store; the most important functions being the support of attributes [17] and relationships [11]. Tedd [18] notes that with a database:

(i) tools need not know how the information is represented
(ii) extra (types of) information can be added without affecting existing tools
(iii) the database structure remains constant through its knowledge of viable relationships. Flexible attributes and relationships can be constructed
(iv) the database offers a transaction mechanism to ensure homogeneous interfaces between tools
(v) the database aids areas like inheritance of information (it lends itself well to version control etc.)
(vi) tools using the database structure are therefore integrated

Following the release of the Stoneman document, doubts were raised about the APSE approach. Druffel [19] expresses concern and explains that a software development methodology, complete with management practices (which in turn are supported by automated tools) should be specifically designed; rather than allow such a methodology to be defined by a collection of available tools. This lead to the Methodman document [20] being released which puts a perspective on the APSE as regards software engineering. Buxton [21] in his summary of Stoneman identifies the salient points of an APSE as being:

(i) database — information repository for entire life cycle
(ii) communication interface — both user and system interfaces
(iii) toolset — integrated set of tools for entire life cycle support

and its aim being that of ease of portability of user programs and software tools; this being achieved by the KAPSE and MAPSE.

4. THE UNIX APPROACH

If the APSE is considered to be a set of inter-related tools, then UNIX is more of a collection of independent tools. Under UNIX the tool interface is not well defined. Files have no type or internal data structure (Kernighan [22]) and hence can be readily passed between tools. The basic system interface is for input and output to treat files, I/O devices and programs alike. Tools interact in a limited way and require a powerful command language in order to construct higher level tools. Owners of files have complete access and therefore a software tool cannot assume any particular internal structure. However UNIX supports and encourages the development and usage of private tools and the combination of tools. Part of the UNIX philosophy is in 'thinking small' (which raises the question of whether or not a UNIX based APSE system would be capable of handling typical large Ada projects), and users/programmers are encouraged to develop tools which need to be written in a specific language by initially creating a 'shell' program. This requires no compilation and can be readily modified. An example of a control language interpreter which is

Ada based but with no 'shell programming' facilities is the Karlsruhe Ada Environment [23]. UNIX also supports I/O re-direction, pipes and background facilities, but although pipes allow modularity, they hide the data-type being passed between the modules.

The UNIX system has become one of the computer market's standard operating systems. Nevertheless, one of the problems with UNIX, as Richie [24] points out, is that there is no unique version of the system. It has evolved by new functions being added, and various organisations adding facilities to meet their own needs. This means that although similar environments exist under UNIX, subtle differences between such versions can cause portability problems. Kernighan [25] points out that with market dominance has come responsibility and the need for an increasing number of 'features' to be provided by computing systems. This need for new ideas has led to creeping featurism, and as a result, the kernel has grown in size by a factor of ten in the past decade (although it has certainly not improved by the same amount!). The whole complexity of dynamic versions of the system is currently being controlled by AT & T in their plan to issue controlled releases (e.g. System III, V ...). Johnson [26] illustrates that, theoretically, it is possible to port software developed in C, and indeed tools have been developed to check portability (lint being one example). However, Norman [27] points out that failures do occur due to inconsistency, a classic example is the difference between the Bourne and C shells.

Law [28] argues that some of the important missing features of UNIX are concerned with maintaining operational efficiency. As the aim of UNIX is for program construction rather than program execution, omissions include: no way to specify at run time the maximum size of a disk file. Also, files get split into many chunks on a fairly ad hoc basis; the result being that disk packs can end up with data distributed in a very fragmented way. The development of the Programmer's Workbench does not solve all these problems, however it does have additional facilities which enables the 'UNIX machine' to provide a development environment for programs which are to be run on other machines. In Ivie's article [29] on the PWB the suggestion is made that a workbench, used in improving the development process, focuses attention on the need for adequate tools and procedures; it also serves as a mechanism for integrating tools into a coordinated set. The workbench adds stability to the programming environment by separating tools from the product. The workbench capabilities include:

(i) system specification and functional description

(ii) system design

(iii) implementation

(iv) testing

(v) conversion - 'going live'

(vi) operation, support and maintenance

It is made up of the following components:

(i) remote job entry - transmitting jobs to target systems and returning output to appropriate users

(ii) module control - where modules are text (files of source code, documentation, data or any other text)

(iii) version control - implemented by means of the Source Code Control System (Rochkind [30]) which records every change made to a module and can then recreate a module as it existed at any point in time. It also controls and manages any number of concurrently existing versions of a module, and offers various audit and administrative facilities

(iv) documentation production

(v) test drivers

Dolotta [31] explains how a convenient working environment and a uniform set of programming tools for use by a very diverse group of users is achieved via the following characteristics:

(i) interfaces built in close co-operation between devices and users

(ii) reliable for production software

(iii) large number of simple, understandable program development tools that can be combined in a variety of ways - users 'package' these tools to create their own specialised environments

5. IS UNIX AN APSE?

APSE is the state of the art, however it is expensive. Pascal played an important role in the development of Ada. It is quite possible that because of its ease of use and extreme popularity, UNIX will play an important role in the development of the APSE. Mitze [32] however points out that UNIX systems do not have requirements analysis and specification, quality assurance, maintenance or support specific software development methodologies. Wegner [33] also states problems with trying to rename UNIX as an APSE or IPSE. The hallmarks of UNIX are its flexibility and user responsiveness, although these are achieved sometimes at the expense of efficiency. It is also convenient to create specialised environments for particular classes of applications, perhaps the most recognised environment being PWB/UNIX [29].

The principles of host/target machine development are exemplified by the UNIX Programmer's Workbench (PWB/UNIX). This has been created for large applications to be developed in a host computer which may be different from the target computer on which the applications are to be run. The PWB/UNIX was designed for program development computers to be medium-sized minis, the target being a faster larger computer. In contrast, real-time computing is generally concerned with using fast, large host computers for developing programs to run on small embedded target computers with limited resources. The implications of this is that although the size of host/target machines is relevant, the functionality of each is quite different in the two situations. Real-time environments must emphasise efficiency and reliability. Tool can efficiency conflict with flexibility; in UNIX large tools are created from small ones. However, it is possible to gain greater efficiency in creating single large monolithic tools, as the interfaces between tool components have been hand-optimised.

UNIX does not use a database which is one of the requirements of Stoneman. There is a need, that UNIX presently does not satisfy, to handle multi-component systems with real-time requirements and a need to simulate target computers under harsh conditions in the host (program development) environment. UNIX has weak multi-tasking and synchronisation primitives. These primitives may have to be replaced with a handler of both shared variables in block-structured environments and distributed processing; this can be achieved by modifying the UNIX kernel. One view of the database is that it replaces the conventional filing system (Hall [34]), and contains, just as a filing system would, all the data of the installation. A database is structured and no filing system comes close to meeting that definition. The key to this abstraction is to separate the data structure from the processes that use it, and to capture this abstraction in a visible way. The data is then capable of being shared between any number of independent tools, each of which need know nothing of the others.

Toolset standardisation (as defined by Glass [35]) is one of the key issues of the APSE, however there has not yet been a commonly accepted definition of what a toolset should contain. The intermediate code, DIANA, provides an interface for a number of software tools - formatters, language-orientated editors etc. Burns [36] looks at the requirements for user interfaces in Ada. Elzer [37] pinpoints the problems of not having a standard internal interface. The absence of a homogeneous tool interface makes the use of tools more error-prone and less efficient, it also prevents large subsets of existing tools from ever being used at all by a reasonably large number of developers working in a particular environment. A homogeneous and structured interface is a key to the expandability of an environment. For example, to build a new feature into a tool or to replace one tool by another which retains the properties of the user interface will make the change transparent to the user. One of the difficulties of producing a consistent toolset is that because the internal interface or database is less visible it has become neglected, whereas tools themselves are fun to build and are self contained - thus there is no lack of them!

In the design of a software development environment special emphasis should be placed on documentation maintenance and re-targetability. They should be easily re-hostable and additionally support design, modelling and simulation. The inter-tool interface can be defined as part of the KAPSE or outside the KAPSE. As Stenning [38] points out though, if it is defined inside the KAPSE, then tools can be transported independently to and from compatible KAPSEs, otherwise they will need local support at the new host for their interfaces; this could be achieved by enclosing the tool within an interface conversion layer.

6. APSE/UNIX IMPLEMENTATIONS

One database management system that has been implemented successfully is the Ada Language System (ALS) Wolfe [39]. This uses a structure of nodes, related as parent and offspring (the tree structure upon which UNIX is based). To provide a common interface for the exchange of tools it uses DIANA. The first KAPSE for the Ada Language System (ALS) was based upon UNIX - Thall [40]. UNIX was chosen because of its relative simplicity, is well received by its user community and has a good record of re-hostability. Not only have entire UNIX systems moved between computers, but UNIX like services have been built on top of well over 50 widely differing operating systems.

Another major APSE/UNIX project is the Project Development Environment (Crowe [17, 41]). This is an attempt to support the Common APSE Interface Set (CAIS) from UNIX. It uses a file hierarchy analogous to the UNIX file system. All UNIX tools are available, new tools have been added for version control and project management. CAIS is designed to promote source-level portability of Ada programs. The CAIS (KIT/KITIA [42]) implementation acts as a manager for a set of entities that may be files, processes or devices. The model uses the notation of a node as a carrier of information about an entity. Initially it has been directed to the interface of the Ada Integrated Environment (AIE) and ALS. Tools written in Ada using CAIS are defined as packages and should be transportable to other CAIS implementations; there may however be problems is porting individual tools, rather than a set. Finally, it is possible to derive an AIE sharing many of UNIX basic facilities, although the two systems differ in file structure, command processor and compiler implementation, Ryer [43]. To support separate compilation with interface type checking, the Ada compiler must maintain a program library. The module interdependencies are part of the source code and are reflected in the program library, rather than being maintained by a separate mechanism, such as the UNIX 'make' program. However, it is possible to use make if a preprocessor distinguishes the dependencies, for indeed, York's adadep [44] does just this, and it can be extended, as in adamake [45], to function in a very similar way to 'make' when used with C programs.

Matrix of Ada Environment Implementations
(Updated November 1983)*

Organisation	Scope	Host System	Target System
Carnegie-Mellon Univ.	Full compiler and programming environment	DEC VAX (UNIX) Three Rivers Computer Corp. PERQ (ACCENT)	PERQ (Also VAX UNIX with post programmer)
Control Data Corp.	Full Ada with partial environment	CYBER-170(NOS)	CYBER-170
Intermetrics Inc. (AIE)	Full Ada plus environemnt ('Stoneman')	IBM 370	same
ROLM Corp.	Full ANSI ADA with toolset	MSE/800 Data General 32-bit ECLIPSE	MSE/800,MSE/14 1666B
SofTech Inc. (ALS)	Comprehensive APSE with 'Stoneman' and including production ANSI Ada compiler	VAX/VMS	same MCF
Olivetti/Danish Datamatics Centre Christ-ian Rovsing Ltd	Full Ada compiler plus Stoneman environment	Christian Rovsing CR880 Olivetti S6000	same
University of Hamburg	APSE implemen-tation planned, using Karlsruhe front end	DEC-10 (Pascal first, Ada-0 super set later)	German minicomputers
Paisley College	CAIS implemen-tation for APSE	VAX/UNIX	same PERQ
University of Bradford	APSE implemen-tation planned using CAIS	VAX/UNIX	same PERQ

Table 2 Taken from [46]

7. CONCLUSION

The aims of UNIX and the APSE have been compared. An APSE is a coordinated set of software tools built around a common database whereas UNIX is more a collection of tools acting upon unstructured files. However, UNIX is widely becoming the de-facto operating system, whereas the APSE is, at the moment, a theoretical programming environment. It would not be wise to 'patch' UNIX to try and make it simulate an APSE. It is, however, quite feasible to take the ideas from behind the APSE, and the design principles of UNIX (i.e. the flexibility, ease of use of tools) and combine them to form a general integrated project support environment using UNIX as a model. This work is subject to development at the moment, with Alvey [47] supporting a #4.5 million IPSE project.

Version control and a flexible attribute model for files are clearly required in any IPSE. A database structure is desirable although a common set of tools (for resource control, life cycle management etc.) integrated by some appropriate workbench shell will provide a cost effective

alternative. The CAIS project has illustrated the use of user controlled file attributes and FACS [11] has enhanced SCCS to provide a comprehensive method of creating and controlling the relationships between files. Structured tools, group project work and life cycle support necessitates that some of the freedom and flexibility associated with UNIX in some situations is limited. Control must shift from the tool users to the tool itself. Files, if they must be used, should become more inconspicuous. Uses must be encourage to focus attention on tools not files. For example, the front end of an Ada compiler may produce a DIANA representation of the source program. A pretty printer will then re-construct an equivalent program. To do this however it must not be possible for the user to corrupt the DIANA form. Thus tools such as the editor must refuse access to a file with certain attributes even to the owner of the file. These restrictions should not be seen as imposing unwanted harness on the programmers. Rather, by allowing the programmer to choose to restrict him/herself then it can be argued that this increases flexibility.

Acknowledgements

This work has been supported by a research studentship from the SERC.

References

1. Ince D., "Software Support: Not Just Patchwork.," *Computing.*, (12 April 1984).
2. Elzer P. F., "The Evolution of the Requirements for the Software Environment for Ada.," *Real-Time Data Handling and Process Control - Proceedings of the First European Symposium.*, pp. 397-401 North-Holland, Amsterdam, 1980, (23-25 October 1979). Berlin (West).
3. Buxton J.N., Druffel L.E., and Standish T., "Recollections on the History of Ada Environments.," *Ada Letters.* **1**(1) pp. 16-21 (1981).
4. Kramer J., "Ada Status and Outlook.," *AGARD Confernece Proceedings No. 330. Software for Avionic (AGARD-CP-330).*, pp. 14/1-14/6 Agard. Neuilly-sur-Seine, France 1983, (6-10 Sept 1982). The Hague - Kijkduin, Netherlands.
5. Department of Defense, *Requirements for Ada Programming Support Environments "Stoneman",* DoD (February 1980).
6. Wasserman A.I., "The Ecology of Software Development Environments.," pp. 47-52 in *Tutorial: Software Development Environments.*, ed. Wasserman A.I., IEEE Computer Society, New York (August 1981).
7. Notkin D.S. and Habermann A.N., "Software Development Environment Issues as Related to Ada.," pp. 107-137 in *Tutorial: Software Development Environments.*, ed. Wasserman A.I., IEEE Computer Society, New York (August 1981).
8. Wasserman A.I., "Towards Integrated Software Development Environments.," pp. 15-35 in *Tutorial: Software Development Environments.*, ed. Wasserman A.I., IEEE Computer Society, New York (August 1981).
9. Osterweil L., "Software Environments Research: Directions for the Next Five Years," *Computer* **14**(4) pp. 35-43 (April 1981).
10. McDermid J. and Piphen K., *Life Cycle Support in the Ada Environment,* Cambridge University Press, Cambridge (1984).
11. Carfagno J., "Using UNIX on a Large Software Project.," *Proceedings of EUUG Conference in Nijmegen.*, (16 April 1984).
12. Bowles K.L., "The impact of Ada on software engineering," *AFIPS Conference Proceedings 1982 National Computer Conference*, pp. 327-332 (7-10 June 1982). Houston, Texas, USA.
13. Kernighan B.W. and Plauger P.J., *Software Tools.*, Addison-Wesley Publishing Company, Reading, Massachusetts (1976).

14. Wasserman A.I., "Software Tools and the User Software Engineering Project.," pp. 93-113 in *Software Development Tools.*, ed. Riddle W.E. and Fairley R.E., Springer-Verlag, Berlin Heidelberg (1980).
15. Buxton J. and Druffel L., "Requirements for an Ada Programming Support Environment: Rationale for Stoneman," pp. 319-330 in *Software Engineering Environments*, ed. H. Hunke, North-Holland ().
16. Goos G., Wulf W.A. (Eds), *DIANA Reference Manual,* Institut Fuer Informatik, University of Kalsruhe (1981).
17. Crowe M.K., Jenkins D.G., MacKay D.N., Ball G.R., and Hughes M., "The Project Development Environment.," Working Document, Paisley College Of Technology - Software Tools Research Group, Paisley (1984).
18. Tedd M. D., "The APSE, ADA Programming Support Environment.," *An Introduction to Ada - A Real-Time Language for Engineers.*, pp. 3/1-3/4 IEE. London, (19 May 1983).
19. Druffel L. E., "Ada Programming Support Environment (APSE).," *Sigsoft Software Eng. Not. (USA).* **7**(2) p. 40 (April 1982).
20. Wasserman A. I. and Freeman P., "Ada Methodologies: Concepts and Requirements.," *Sigsoft Software Eng. Not. (USA).* **8**(1) pp. 33-50 (Jan 1983).
21. Buxton J.N., "The Stoneman Project and Summary Systems for Ada: Summary," *Information Technology for the Eighties, Proceedings BCS'81 Conference London*, p. 1 Heyden, London, 1981, (1-3 July 1981).
22. Kernigham B.W. and Mashey J.R., "The Unix Programming Environment," *Software - Practice and Experience* **9** pp. 1-15 (1979).
23. Dausmann M., Jansohn H-S., Kirchgassner W., Landwehr R., Persch G., and Uhl J., *Karlsruhe Ada Environment, User Manual,* Institut fur Informatick II, University Karlsruhe ().
24. Richie D. M. and Thompson K., "The UNIX Time-Sharing System.," *The Bell System Technical Journal.* **57**(6) pp. 1905-1929 (July-August 1978).
25. Kernighan B.W. and Pike R., *The UNIX Programming Environment.,* Prentice-Hall Inc., Englewood Cliffs, New Jersey (1984).
26. Johnson S. C. and Ritchie D. M., "Portablility of C Programs and the UNIX System.," *The Bell System Technical Journal.* **57**(6) pp. 2021-2048 (July-August 1978).
27. Norman P.A., "The Trouble with UNIX.," *Datamation*, pp. 139-150 (November 1981).
28. Law D.T. and Abbott J., *Programmer Workbenches,* NCC Publications, Manchester (1983).
29. Ivie E.L., "The Programmer's Workbench - A machine for Software Development.," *Communications of the ACM.* **20**(10) pp. 746-753 (October 1977).
30. Rochkind M.J., "The Source Code Control System.," *IEEE Transactions on Software Engineering.* **1**(4) pp. 364-370 (December 1975).
31. Dolotta T. A., Haight R. C., and Mashey J. R., "The Programmer's Workbench.," *The Bell System Technical Journal.* **57**(6) pp. 2177-2201 (July-August 1978).
32. Mitze R., "Unix as a Software Engineering Environment," pp. 345-357 in *Software Engineering Environments*, ed. H. Hunke, North-Holland ().
33. Wegner P., "The Ada Language and Environment," *SIGSOFT* **5**(2) pp. 8-14 (1980).
34. Hall J. A., "Database Issues in Program Support Environments.," *Ada UK News* **5**(1) pp. 29-38 (Jan 1984).
35. Glass R.L., "Recommended: A Minimum Standard Software Toolset," *SIGSOFT Software Engineering Notices (USA)* **7**(4) pp. 3-13 (October 1982).

36. Burns A. and Robinson J., *Tool Requirements for the Implementation of User Interfaces in Ada,* Ada U.K. News (1984).

37. Elzer P.F., "Some Observations Concerning Existing Software Environments," pp. 227-260 in *Part 1, Minutes 7th Annual Meeting International Purdue Workshop on Industrial Computer Systems*, (8-11 October 1979). West Lafayette, IN, USA.

38. Stenning V., Froggatt T., Gilbert R., and Thomas E., "The Ada Environment: A Perspective," *Computer* **14**(6) pp. 26-36 (June 1981).

39. Wolfe M., Babich W., Simpson R., Thall R., and Weissman L., "The Ada Language System," *Computer* **14**(6) pp. 37-45 (June 1981).

40. Thall R.M., "The KAPSE For The Ada Language System," *Proceedings of AdaTEC Conference on ADA*, pp. 31-47 ACM, (6-8 October 1982). Arlington, Virginia, USA.

41. Crowe M. K., Ball G., Hughes M., Jenkins D., and Mackay D., "Supporting the CAIS from UNIX.," *Ada UK News* **5**(1) pp. 48-50 (Jan 1984).

42. KIT/KITIA CAIS Working Group for AJPO, *Draft Specification of the Common APSE Interface Set (CAIS) Version 1.1.*. 30 September 1983.

43. Ryer M., "Developing an Ada Programming Support Environment," *Mini-Micro Systems*, (No. 10) pp. 223-225 (1982).

44. Briggs J., *adadep,* University of York, York (23 September 1983).

45. Morrison I.W., *adamake,* University of Bradford, Bradford (20 February 1984).

46. Ada Letters, "Matrix of Ada Language Implementation.," *Ada Letters* **3**(4) pp. 152-157 (Jan, Feb 1984).

47. Talbot D. and Witty R. W., *Alvey Programme - Software Engineering Strategy.*, Alvey Directorate, London (November 1983).