AUUGN

Australian Unix systems User Group Newsletter

Volume 8

Number 5

Includes Winter 1987 Conference Papers

Registered by Australia Post Publication No. NBG6524

The Australian UNIX* systems User Group Newsletter

Volume 8 Number 5

October 1987

CONTENTS

* UNIX is a registered trademark of AT&T in the USA and other countries.

AUUG General Information

Memberships and Subscriptions

Membership, Change of Address, and Subscription forms can be found at the end of this issue.

All correspondence concerning membership of the AUUG should be addressed to:-

The AUUG Membership Secretary,
P.O. Box 366,
Kensington, N.S.W. 2033.
AUSTRALIA

General Correspondence

All other correspondence for the AUUG should be addressed to:-

The AUUG Secretary,
Department of Computer Science,
Melbourne University,
Parkville, Victoria 3052.
AUSTRALIA

ACSnet: auug@munnari.oz

AUUG Executive

Ken McDonell, *President*

kenj@moncsbruce.oz
Department of Computer Science, Monash University, Victoria

Robert Elz, *Secretary*

kre@munnari.oz
Department of Computer Science, University of Melbourne, Victoria

Chris Maltby, *Treasurer*

chris@gris.oz
Softway Pty. Ltd., N.S.W.

Chris Campbell, *Committee Member*

chris@olisyd.oz
Olivetti Australia, N.S.W.

Piers Lauder, *Committee Member*

piers@basser.cs.su.oz
Basser Department of Computer Science, Sydney University, N.S.W.

John Lions, *Committee Member*

johnl@elecvax.oz
School of Electrical Engineering and Computer Science, University of New South Wales, N.S.W.

Tim Roper, *Committee Member*

timr@labtam.oz
Labtam Limited, Victoria

Next AUUG Meeting

The next meeting will be held in Melbourne during August-September 1988.
Futher details will be provided in the next issue.

AUUG Newsletter

Editorial

I hope you enjoy this issue and please contribute to the next issue.

REMEMBER, if the mailing label that comes with this issue is highlighted, it is time to renew your AUUG membership.

AUUGN Correspondence

All correspondence reguarding the AUUGN should be addressed to:-

John Carey
AUUGN Editor
Computer Centre.
Monash University
Clayton, Victoria 3168
AUSTRALIA

ACSnet: auugn@monu1.oz

Phone: +61 3 565 4754

Contributions

The Newsletter is published approximately every two months. The deadline for contributions for the next issue is Friday the 11th of December 1987.

Contributions should be sent to the Editor at the above address.

I prefer documents sent to me by via electronic mail and formatted using *troff -mm* and my footer macros, troff using any of the standard macro and preprocessor packages (-ms, -me, -mm, pic, tbl, eqn) as well TeX, and LaTeX will be accepted.

Hardcopy submissions should be on A4 with 35 mm left at the top and bottom so that the AUUGN footers can be pasted on to the page. Small page numbers printed in the footer area would help.

Advertising

Advertisements for the AUUG are welcome. They must be submitted on an A4 page. No partial page advertisements will be accepted. The current rate is AUD$ 200 dollars per page.

Mailing Lists

For the purchase of the AUUGN mailing list, please contact Chris Maltby.

Disclaimer

Opinions expressed by authors and reviewers are not necessarily those of the Australian UNIX systems User Group, its Newsletter or its editorial committee.

President's Report

This issue of AUUGN contains most of the papers presented at the recent Winter Technical Meeting held at NSWIT, August 27-28.

That meeting, and the "up-market" approach to the associated exhibition was the result of some very hard work and innovation on the part of the local organizing committee under Greg Webb's chairmanship. Once again I'd like to thank them for an excellent job, well done.

Concurrent with the Sydney technical meeting was an AUUG management committee meeting at which the following resolutions were made,

- There will be a serious attempt to increase benefits that flow from AUUG membership, particularly in respect of redistribution of software and publications from affiliated user groups.
- A second round of 4.3BSD manual orders will be processed; the order form will appear in the next AUUGN.
- AUUG should actively solicit support from vendors of Unix-related products, especially in respect of publicizing the group and attracting prospective new members.
- The AUUGN editor shall henceforth be invited to attend management committee meetings to be involved in discussions relating to publications and to prepare a report on the meeting for the next AUUG (prior to publication of the minutes of the meeting).
- AUUG should take a more active role in soliciting financial support and more equitable cost-sharing arrangements to guarantee the long-term viability of ACSnet.
- Matters of policy relating to meeting organization (dinner costing, sponsorship, credit card facilities, student participation, etc.) will be formalized and collected into a document that will be given to future meeting organizing committees.

Long-term planning of activities, including the venue, schedule and format of technical meetings is a matter requiring our urgent attention. At this stage there seems to be considerable support for a NSWIT-like meeting held once per year (probably in August or September); this leaves the format of the other (summer) meeting unresolved. At this stage, convergence on a rational short-term schedule may demand that no national summer meeting be held in 1988, but a revamped and professionally organized winter meeting with full-scale equipment exhibition would be held in Melbourne around early September 1988.

As a substitute for a national 1988 summer meeting, there is possibility that AUUG could sponsor an overseas speaker(s) tour in February, or encourage several smaller State-based informal technical meetings in the same time frame.

Given the importance of this departure from historical precedent, and the fact that the matter cannot be finally resolved until the next management committee meeting on December 10, all members can expect to receive correspondence from AUUG by mid-December outlining the meetings programme for 1988 and beyond.

In the interim, I would welcome your comments either spoken ((03) 565 3899), e-mail (kenj@moncsbruce.oz) or via news (aus.auug).

As this will be my last contribution in this role (I depart for the U.S. soon, and shall be resigning as AUUG President), I would like to take this opportunity to extend my sincere thanks to all members of AUUG for their contributions, and in particular to the members of the management committees and the AUUGN editors.

Ken J. McDonell

Softway

a Techway company

for

- ✓ UNIX System V
- ✓ Documentor's Workbench 2.0
  - and various back-end drivers
  - PostScript support of plain text
  - support for graphs and images
- ✓ Ports & Device Drivers
- ✓ Intelligent Benchmarking
- ✓ SUN-III (ACSnet) + installation
- ✓ Biway - Bi-directional modem software for System V and 4bsd
- ✓ Courses:
  - Beginner's Workshop
  - Fast start to UNIX
  - System Administrators' workshop
- ✓ Technical Backup

- and all sorts of interesting software development.

Softway Pty Ltd. (Incorporated in NSW)
120 Chalmers St, Strawberry Hills, NSW.
PO Box 305, Strawberry Hills, NSW 2012.
☎ (02) 698 2322 Fax (02) 957 6914

Adelaide UNIX Users Group

The Adelaide UNIX Users Group has been meeting on a formal basis for 12 months. Meetings are held on the third Wednesday of each month. To date, all meetings have been held at the University of Adelaide. However, it was recently decided to change the meeting time from noon to 6pm. This has necessitated a change of venue, and, as from April, meetings will be held at the offices of Olivetti Australia.

In addition to disseminating information about new products and network status, time is allocated at each meeting for the raising of specific UNIX related problems and for a brief (15-20 minute) presentation on an area of interest. Listed below is a sampling of recent talks.

D. Jarvis	"The UNIX Literature"
K. Maciunas	"Security"
R. Lamacraft	"UNIX on Micros"
W. Hosking	"Office Automation"
P. Cheney	"Commercial Applications of UNIX"
J. Jarvis	"troff/ditroff"

The mailing list currently numbers 34, with a healthy representation (40%) from commercial enterprises. For further information, contact Dennis Jarvis (dhj@aegir.dmt.oz) on (08) 268 0156.

Dennis Jarvis,
Secretary, AdUUG.

Dennis Jarvis, CSIRO, PO Box 4, Woodville, S.A. 5011, Australia.

PHONE: +61 8 268 0156
UUCP: {decvax,pesnta,vax135}!mulga!aegir.dmt.oz!dhj
ARPA: dhj%aegir.dmt.oz!dhj@seismo.arpa
CSNET: dhj@aegir.dmt.oz

AUUG Winter 1987 Conference Papers

In order of presentation

AUUG Winter 1987 Conference Papers

continued

UNIX at the Turn of the Century

Michael Tilson
HCR Corporation
130 Bloor Street West, 10th Floor
Toronto, Ontario M5S 1N5 Canada
(416) 922-1937
{utzoo,ihnp4,...}!hcr!mike

UNIX has been available outside Bell Labs since about 1974. Thirteen years ago the system was new, still experimental, and rarely used. Today, UNIX is mature, becoming standardized, and widely used. What can we expect in the next thirteen years? This talk discusses the technology trends that will determine the status of UNIX at the turn of the century.[1]

UNIX has become a standard working environment for software development. The lifetime of standards is surprisingly long. FORTRAN has been with us for a long time, and it looks like it will be with us for decades to come. Today's UNIX system will still work fine until at least late January, 2038.[2]

On the other hand, technology continues to advance at a rapid rate. Systems that once appeared modern become obsolete and obstacles to productivity. There is no reason to believe that the rate of change will slow between now and the end of the century. The important trends that must be considered include memory sizes, processor speed, network bandwidth, networking and communications software, user interface hardware and software, and software development technologies. We will see low cost, extremely powerful, more friendly computer systems, that have very high bandwidth connections to other systems. UNIX must adapt to these changes.

The existence of virtually identical software environments on almost all machine architectures opens up possibilities that never before existed. The multi-vendor NFS demos that now occur at many UNIX commercial exhibitions would have been unthinkable not very long ago. But in the next thirteen years UNIX will open the door to possibilities for distributed processing and distributed applications that go far beyond anything we can do today.

This talk attempts to reconcile the conflict between the pressing need for change and the inertia of standards. A technical forecast is provided, giving a framework for looking at UNIX systems development over the next decade. The goal is to understand why a typical obsolete C application written in the mid-80's might be still running on an incredibly advanced architecture, moving data from New York to Tokyo in the year 2000.

Forecasting for the next millenium is a dangerous business. Historically, the advent of a new millenium triggered a plethora of forecasts. The talk will touch on some of the interesting parallels with events that occurred around the year 1000.

[Note: A shorter version of this talk was given at the Usenix Conference, June 1987.]

---

[1]The pedantic reader will notice that the turn of the century is assumed to be the year 2000, and of course this really happens January 1, 2001. However, I suspect that when the time comes, the big celebration (or the wait for the end of the world) will come a year earlier. Anyway, UNIX programmers prefer 0-indexing.

[2]On 32-bit processors the current UNIX time algorithms will overflow after this date. Still, this is it quite a bit better than some other systems that will fall over dead after December 31, 1999. When 64-bit processors become the norm, future timekeeping may be restricted only by limitations of storage needed to hold the time zone and daylight savings algorithms.

The Locking of Critical Regions Under UNIX™

Frank Crawford

Q.H. Tours
PO 630, North Sydney 2060

and

Jagoda Crawford

Australian Nuclear Science and Technology Organisation
Private Mailbag 1, Menai 2234

ABSTRACT

As more multiprocessor systems running UNIX come into common use, there is a need to re-examine the standard techniques employed to lock critical regions in concurrent processes. This paper identifies some of the inadequacies of these methods and details a number of functions available in AT&T's System V and/or Berkeley's 4.2/4.3 BSD that give secure locking in both single and multiprocessor environments. Finally, an example is given of a method suitable only for a single processor environment and a corresponding method for multiprocessors.

™ UNIX is a registered trademark of AT&T in the USA and other countries.

1. Introduction

To meet the continuing demand from science and industry, manufacturers are seeking new methods for realising more computer power. One method that has emerged recently is to utilise a number of separate processors in the one system, *i.e.* multiprocessor systems. In all current commercial applications, the aim of multiprocessor systems is to increase throughput rather than decrease the execution time of a single process.

There are two broad categories of multiprocessing. One is to dedicate all available processors to the execution of a single job, dividing it into a number of separate *threads*, each of which is run on a separate processor with the system handling synchronisation and communication. This is still a very active research area and is not yet available for commercial use. The second approach is much coarser, each job being run on a separate processor, with no interaction between jobs.

The simple view of multiprocessing given above does not describe the whole picture. When a modern operating system supporting multiprogramming is introduced, additional complexities arise. As an example, it is not desirable to tie a given process to a particular processor; rather it must be capable of being switched between any of the available processors. Further, the scheduling algorithm must be able to handle a number of jobs concurrently.

Despite the additional complexity, multiprocessors have a number of advantages over a number of separate processors, mainly in the sharing of resources, *e.g.* memory, peripherals and especially sharing the workload. This sharing of resources has long been a feature of multiprogramming and many of the principles can be applied to multiprocessing. In effect, multiprocessors are a development of multiprogramming.

2. Synchronisation

When resources are shared by a number of processes, it is possible that the same resource, *e.g.* tape drives, shared memory or files, will be required by more than one process at a given time. Care must be taken to ensure that access to any one of these resources is serialised. Serialisation may be handled directly by the operating system (*e.g.* for tape drives) or left to the processes to synchronise their activities. The type of resource generally dictates at which level the synchronisation is done, *i.e.* kernel or process. This paper only considers synchronisation at the process level and, in general, ignores how the kernel may handle it.

To take a more concrete example, access to shared memory is considered, although it would be just as easy to consider access to a file (*e.g.* a database application) or to some other device.

2.1 Simple Locking Scheme

One of the simplest means of locking is to put aside a single variable, called **lock**, and to set it to one to lock the resource. Similarly, set **lock** to zero to unlock the resource. The scheme in Figure 1 shows a simple function, for this purpose[1].

1. Examples in this paper show only relevant parts, no attention being given to dealing with error conditions, *etc.*

```
extern int lock;

void lock_resource ()
{
        while (lock != 0)
                /* Wait */ ;
        lock++;
}

void unlock_resource ()
{
        lock = 0;
}
```

Figure 1. Simple Locking Scheme.

This is the most common form of locking used, and on single processor systems it works well. However, it may fail in a multiprocessor environment, and further it can even fail in a single processor environment. The reason for this failure is fairly simple to explain; if two processes are using this scheme to lock a resource, it is possible that one of the processes will be swapped between finding **lock** to be zero and setting it, during which time the other process may well execute the same code, and so both may gain access.

This procedure can be modified to work on a single processor system by making use of two or more variables. However, on a multiprocessor system it can still fail. This is more difficult to explain, but the basic problem is that there is no indivisible instructions available to the programmer, unlike the ++ operation on a single processor system[2].

On a multiprocessor system it would be possible for two (or more) processes to be executing the same instructions at the same time, so that both attempt to increment the **lock** variable at the *same* time. More realistically the memory read and write cycle would mean that only one increment would be effective, overiding the other one.

2.2 Theoretical Considerations

Synchronisation problems, such as those mentioned above, have been studied extensively since the mid 1960s, and a number of solutions proposed. The sequence of statements that *must* appear to be executed as an indivisible operation, such as the **lock_resource** and **unlock_resource** functions above, is called a *critical section.* The term *mutual exclusion* refers to the fact that only one process can be in its (common) critical section at any time. Mutual exclusion is used to refer to shared objects (*e.g.* data structures, files, *etc.*) whereas critical sections refer to process segments.

The proposed solutions can be grouped into two broad categories: one where the programmer controls all of the details of the exclusion, and the second, where a set of primitives or programming constructs are provided which hide the details.

2.2.1 User controlled synchronisation Even within this category there are two separate areas: the low level (hardware) method, *e.g.* locking the bus, disabling interrupts and test and set instruction, all of which may be used within a kernel, but are not directly available to user level programs; and higher level algorithms such as Dekker's algorithm [Dijkstra 1965] and others. All of the software proposals require a knowledge of how many processes are going to be sharing the resource, something which is not generally known beforehand in a multiprogramming environment. They also suffer from the

2. This is not guaranteed but is commonly assumed, as any useful computer architecture should have an increment instruction, whether on a register or a memory location. Similar arguments apply to a clear instruction.

drawback of requiring a *busy-wait,* which is wasteful of processor time. Thus these methods are not suited to the average computing environment.

2.2.2 System supplied constructs In all of these methods the system hides the details of implementation within the kernel, and provides various calls for the programmer. These methods include *semaphores, monitors* and *message passing.* These can all be shown to be equivalent, *i.e.* each one can be implemented in terms of another. For a more detailed discussion see Tanenbaum [1987].

From this brief discussion, it can be seen that for a programmer to make use of any of the synchronisation methods, it is necessary for the construct to be built into the kernel. This is the case with the two most common versions of UNIX, AT&T's System V and 4.2/4.3BSD.

3. General UNIX Façilities

Although Edition 7 and earlier versions of UNIX did not offer explicit methods of locking regions, there were a number of features which could be utilised to achieve locking. *Creat* can be used to create a file with mode 0 (or anything without write permission). If this fails, repeat the process until successful, *i.e.* a busy wait. When processing is finished, remove the file. Using a lock file, the previous example can be rewritten as follows:

```
#define LOCKFILE "lock"

void lock_resource ()
{
        int fd;

        while ((fd = creat(LOCKFILE, 0)) < 0)
                /* Wait */ ;
        (void) close(fd);
}

void unlock_resource ()
{
        (void) unlink(LOCKFILE);
}
```

Figure 2. Locking using *creat.*

The example in Figure 2 relies on the use of permissions to stop the creation of an existing file. However this procedure fails for **root**, who commonly needs to perform some form of locking. It also has the side effect of creating a number of files which are not required, and the possibility of leaving a lock after the process has died.

A number of schemes relying on the linking of files can also be used, but these suffer from problems such as race conditions within the kernel. On these versions of UNIX it is not possible, in general to implement a reliable locking scheme.

4. 4.2BSD Facilities

Berkeley Software Distributions[3] 4.2 (and 4.3) offer a number of methods of concurrency control, the majority of which again make use of the file system.

3. Berkeley Software Distribution was developed from Edition 7 UNIX and is distributed by the University of California, Berkeley.

The simplest of the methods enhances the method shown in Figure 2. The main change is the ability to specify that file creation is to fail if it already exists (*i.e.* specifying (O_CREAT|O_EXCL) to *open*). Again all these methods entail busy waiting.

Another method using the file system is to use the file locking system call, *flock*. This can be used as a semaphore and is guaranteed to enable concurrency among cooperating process. It enables processes to be blocked awaiting the release of the 'lock', thus not wasting processor time. Also, as the kernel is handling the lock, it is able to remove it automatically if the locking process dies.

Again, the fact that *flock* uses the file system has the possible disadvantage that for concurrency control not involving files, a special file must be created. This means it cannot be implemented across distributed systems without a common file system. For processes using common files, this is a reasonable method. .

A general but more complex method available under 4.2BSD is to use message passing. For processes sharing a common parent it can be as simple as *pipes*, but for those without a common ancestor, it becomes necessary to use *sockets*. The concept of message passing is simple; all processes wanting access to a particular resource send a message to a single locking process or daemon, who's sole function is to serialise entry into critical sections.

Individual processes wishing to enter their critical regions send a request to the locking daemon and wait for a reply. Upon leaving they send another message. This is depicted in Figure 3. This method is used in some database management systems, e.g. Ingres™. An example of this is shown below, however the detail of setting up the connection and the locking daemon are omitted.

```
#define REQUEST          "My Turn?"
#define FINISHED         "Finished"
extern int lock_fd;

void lock_resource ()
{
        char buf;

        (void) write (lock_fd, REQUEST, sizeof(REQUEST));
        (void) read (lock_fd, &buf, sizeof(buf));
}

void unlock_resource ()
{
        (void) write (lock_fd, FINISHED, sizeof(FINISHED));
}
```

Figure 4. Locking Using Message Passing.

5. System V Facilities

System V offers a number of different facilities for handling concurrency and locking. As with 4.2BSD, it is possible to use the file system via *creat* or *open*. It is also possible to use file locking with *fcntl* and/or *lockf*. Although the details are different, the outcome is the same.

Apart from these facilities, System V also offers some direct support for concurrency. These are *semaphores* and *messages*. Again, although the implementation of *messages* is very different from

™ Ingres is a trademark of Relational Technology International.

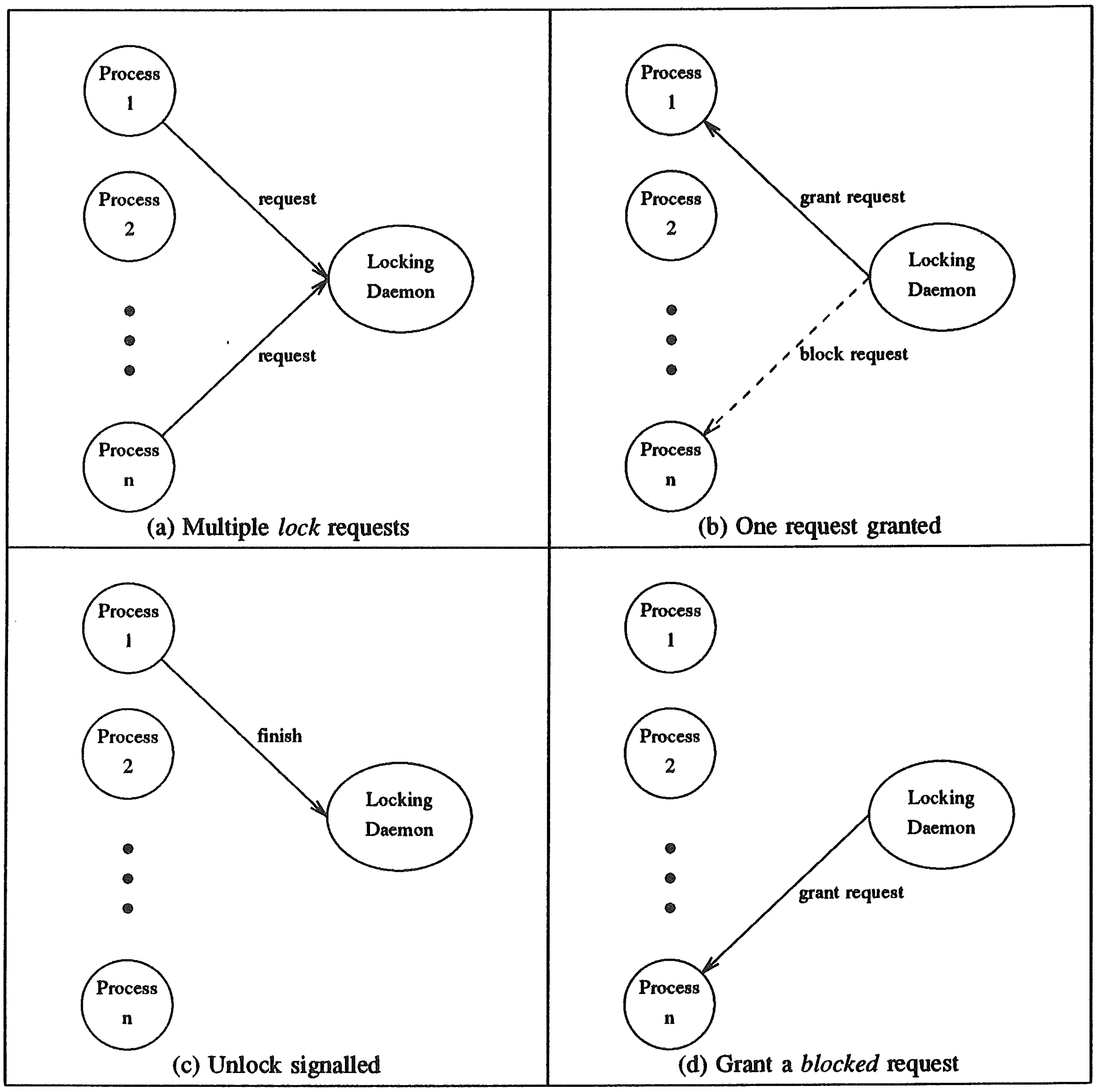


Figure 3. Sequence of Events with Message Passing.

4.2BSD's *sockets*, for the purpose of locking it can be considered the same.

Semaphores are not directly available under any other version of UNIX, and their implementation is similar to standard text book definitions.

5.1 An Example Using Semaphores

Consider an example of a system where there are a number of processes using shared memory, such as a relational database with a per table write through cache. Accessing this cache is a critical section, as any writes must not change the data being read. Further even reading can modify the cache because entries not in the cache will cause it to update.

As there are a number of separate tables, it is best to have a separate semaphore for each table (using only one would degrade performance of the entire system).

The initialisation procedure could be as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define DATABASE        "file.db"       /* The database file */
#define NOTABLES        75              /* Number of database tables */
#define VERSION         1               /* Some version number */
#define SEMPERM         0666            /* Everyone can get it */

extern key_t    ftok ();

extern int      semid;

void sem_init ()
{
        register int    i;
        union semum     arg;

        semid = semget (ftok (DATABASE, (char) VERSION),
                                NOTABLES, (SEMPERM | IPC_CREAT));
        arg.val = 1;                    /* Initial value of semval */
        for (i = 0; i < NOTABLES; i++)
                (void) semctl (semid, i, SETVAL, arg);
}
```

Figure 5. Initialising the Semaphores.

This initialisation sets the maximum number of processes in the critical region to one, by setting arg.val. This is known as a binary semaphore. In other applications it could be set to different values.

After this initialisation, individual semaphores within the set would be used to control access to each table, as follows:

```
extern int      semid;

void lock_resource (table)
        int table;
{
        struct sembuf    sops,
                         *sops_ptr = &sops;

        sops.sem_num = table;
        sops.sem_op = -1
        /* Increment semadj for process exit */
        sops.sem_flg = SEM_UNDO;

        (void) semop (semid, &sops_ptr, 1);
}


void unlock_resource (table)
        int table;
{
        struct sembuf    sops,
                         *sops_ptr = &sops;

        sops.sem_num = table;
        sops.sem_op = 1
        /* Decrement semadj for process exit */
        sops.sem_flg = SEM_UNDO;

        (void) semop (semid, &sops_ptr, 1);
}
```

Figure 6. Locking Using Semaphores.

The functions in Figure 6 can then be used around the critical sections to ensure the integrity of the cache. Further, if a process terminates before unlocking the semaphore, the semaphore is automatically released by the system because the flag **SEM_UNDO** is set in the call to *semop* within **lock_resource**. For a more detailed description of these functions, see the UNIX System Programmer's Manual [1983].

The advantage of using these procedures is that the kernel suspends execution of the calling process if another process is already in its critical region.

6. Conclusions

Although many programs use varying degrees of concurrency, only a few attempt to implement mutual exclusion. In the past, this was because UNIX did not offer many facilities, however with recent versions, various methods have become available. With the growing emphasis on interprocess communication and multiprocessor machines, it is becoming more important for programmers to make use of these facilities to ensure the integrity of shared resources.

References

Dijkstra, E.W. [1965] - Co-operating Sequential Processes. In Programming Languages, *ed.* F.Genuys, Academic Press, London.

Tanenbaum, A.S. [1987] - Operating Systems - Design and Implementation, Prentice-Hall, Englewood Cliffs, New Jersey.

Unix System Programmer's Manual [1983] - AT&T, Murray Hill, New Jersey.

Fun with virtual memory

L. Chubb

University of New South Wales, Kensington, NSW 2033

ABSTRACT

A version of the UNIX version 7 kernel ported to an NS32016 based microcomputer will form the starting point for modifications designed to allow shared memory, incremental loading, the mapping of files into a process's address space and changes to the way shared text is implemented.

The standard version 7 memory management software swapped entire segments between memory and secondary storage. During the port it was changed to use demand paging. Paging is done from a process image file, a regular file corresponding to the process's address space. This gives a one to one mapping between the pages in the address space and the contents of the file.

The proposed changes will allow many files to be mapped into a single process's address space and will allow different processes to use the same file. Processes shareing a file use the same in-memory copies of pages belonging to that file, effectively allowing them to share memory.

Three new system calls have been designed to create, delete and extend mappings between portions of the address space and files. The *fork, exec* and *exit* system calls will also be modified to use these mappings.

1. *Introduction*

On finding myself engaged in porting UNIX* as part of a masters degree I decided it would be a good opportunity to investigate what could be done with virtual memory in UNIX. This paper reviews briefly some of the more interesting functions provided by two implementations of virtual memory in UNIX today and present a few of my own ideas on the subject. The use of virtual memory in UNIX system V with its *regions* and *shared memory* and then UNIX Version 8 with the concept of processes as files will be examined briefly.

My ideas evolved while I was porting the UNIX Version 7 kernel to an NS32016 based microcomputer and contain elements found in the other versions of UNIX. During the port the use of swapping in standard Version 7 was replaced by demand paging from a process image file. Version 7 was ported as it was the only version of the source readily available to me when I started.

By associating each segment of the address space with a file and allowing the user to create and remove segments the process can be debugged by reading and writing the file representing stack and data areas, as in Version 8; and shared memory can be provided, which is a feature of System V. The same mechanism can be used to map any regular file into the process's address space allowing dynamic linking and shared text.

* UNIX is a registered trademark of A.T.&T. Bell Laboratories

2. *Processes under UNIX Version 8*

UNIX Version 8 [Killian 85] provides a new directory called */proc.* Each file in this directory corresponds with the process image of a running process and is used for swapping or paging. This representation appears to have been motivated by the limitations of previous versions of UNIX during interactive debugging where the debugger and the object being debugged are separate processes.

The owner of the process image file is the same as the owner of the process. When the file is created the permissions allow reading and writing for the owner only. The size of the file is the size of the virtual space of the process. A feature particularly useful for debugging is the ability to read and write any part of the process's virtual space using the *read, write* and *lseek* system calls. When a write is performed on a shared text segment a private copy is made. Using these routines to access the files allows them to be protected using the normal file protection mechanisms.

3. *System V*

System V [Bach 86] divides the address space of a process into areas called *regions* containing either text or data. There are two data structures involved in mapping regions. One is a *region table* containing an entry for each active region in the system giving information about where its contents are located in physical memory, a pointer to the inode whose contents were originally read into the region by *exec*, the size, status, region type and number of processes referencing the region. The second is a *pregion table*, or per process region table, containing the virtual starting address of each region, a pointer to the region table and the permissions associated with each region.

The text, data and stack regions are created by *fork* or *exec*, but system calls are also provided for the user to manipulate regions. When *exec* creates the text and data regions the contents of the executable file are read into these regions.

Shared memory is provided when a region is attached to the address spaces of several processes at once. Each shared region has an entry in the shared memory table *shmid_ds* giving such information as the size of the region, the creator's identity, the number of times it is currently attached to an address space, and so on. The shared memory is protected in a manner similar to that of files having read and write permissions for the owner, the group and others. Shared memory must not overlap other regions in the address space. If the size of the region is to be increased its location must be carefully chosen to allow for this expansion.

Before shared memory can be used a new region of shared memory has to be created using a system call *shmget.* It returns a shared memory identifier which uniquely identifies the region. A region specified by the shared memory identifier can be attached to a process's address space by using the *shmat* system call specifying the virtual address where it will reside. If the user does not specify an address the operating system chooses one.

When a process is finished with the shared memory the region can be removed from its address space by using the *shmdt* system call. The region to be detached is identified by the virtual address in case the same region is attached to the address space more than once.

Another system call, *shmctl,* is provided to return information about shared memory, change the user or group id's or the permissions, and remove the shared memory region.

4. *Functions to be provided by my system*

What is discussed here are some building blocks which may be placed within utilities or runtime libraries to provide new functions. The new functions possible are:

1. Shared text.

2. Dynamic linking (or incremental loading) is provided by allowing object files to be mapped into the address space. This would involve providing some runtime library routines to store the address of the object code and transfer control. Giving executable objects a one block header would simplify the matter so that the header need not be mapped into the address space.

3. Shared memory is obtained by mapping a file with read/write permission into the address spaces of more than one process at once.

4. The process image can be accessed by using read or write on the process image file which contains the data and stack segments.

5. A data file can be mapped a data file into a process's address space allowing the file to be used as if it were memory.

The benefit of most of these features is well enough known to forgo any discussion of them here but the ability to map data files into a process's address space deserves further comment. One of the big advantages of accessing files in this way is reducing the complexity of any program needing access to a file, particularly when the access is random. Examples where this type of access may simplify a program are an editor mapping in its work file or an interpreter mapping in the interpreted code. As the usual file handling routines are used by the page fault routine it takes no longer to process a file in this way than using the *read* or *write* system calls.

All the functions mentioned above are provided by the same mechanism which is of benefit in terms of the complexity of the kernel. Overall there may be some price associated with paging but this has yet to be determined.

4.1 *Segments*

Users will notice the effects of this implementation of segments only if they choose to use the new system calls *makeseg, rmseg, exseg* and *segstat* to map files into the address space. The use of process image files by *fork, exec,* and *exit* makes little visible difference except for the existence of the directory */proc* containing the process image files. This makes existing programs compatible with the new system. There is no reason why mapping should be used if not desired as files can still be accessed in the usual way.

The process image is maintained as a normal UNIX file whose size is exactly 16MByte, which is the maximum size of an address space on the NS32016. It contains the data and stack segments. The existing kernel file handling routines are used to access, create and delete the process image files created during a *fork* or *exec.* These files are maintained until the process exits.

Only the data and stack segments created upon execution are kept as part of the process image file. No actual data blocks are allocated for the gap between the data and stack segments which lie at opposite ends of the address space. The gap between them in the process image file allows for the expansion of those segments. When the segments are extended using *exseg* to cover non-existent parts of the file missing blocks are read as nulls by the file system software. An implications of having a gap in the process image file is that the usual file copying procedure cannot be followed when the process image is duplicated during a fork. It would write blocks for each block in the gap wasting large amounts disk space. Process image files are copied by a kernel routine which copies files exactly, preserving any gaps.

The system call *makeseg* will map a file into the address space so that the mapped section of the file appears to be a section of memory containing the information stored in the file. The paging software reads in the block corresponding to that part of the address space when it is accessed, and in a manner invisible to the user. When either the system call *rmseg* is used to remove a mapping or the process exits the file is updated if it is read/write, and closed.

There are costs in time and memory in representing segments in this way. The segment map and shared file array require more memory. More time is needed for paging. When a page is to be supplied it is possible that file index blocks need to be read also in order to find where the required file block is located. Buffer caching may negate this to some extent. If a second level page table is required the list of page tables in the shared file array must be searched before allocating a new page table to ensure it does not exist elsewhere in the system.

4.2 *Data Structures*

A segment, corresponding roughly to a region in System V, is an area of the address space mapped onto a contiguous block within a single file. The mapping of a segment to a file is defined by two data structures, the *segment map* and the *shared file map*. These structures are used when pages are read into memory or released and in the allocation and freeing of second level page tables.

The segment map is an array within the user area. Each segment map entry gives all the information required to map a single segment within the process's virtual space. It includes the virtual address, size, length, and permissions associated with the segment.

The shared file array contains an entry for each file mapped into any process's address space. It maintains a count of the number of processes using that file at any time allowing the entry to be deleted when the count falls to zero. Its main purpose is to keep track of any pages belonging to the file residing in memory, by maintaining a list of the second level page tables which map those pages and a count of the number of processes using each table.

There is a cost in maintaining a shared file entry for files that are not being shared. The maximum cost will be borne if all files are always shareable. Savings may be made either if files are assumed to be shareable unless otherwise stated or if they are assumed to be non-shareable unless stated. Sharing a file involves extra processing when paging and requires memory to store information on the file. However, treating all segments in the same way does simplify the kernel code.

4.3 *Paging*

The memory management hardware requires each process to have its own first level page table and set of second level tables used to translate virtual addresses. Processes share pages in memory by sharing the second level page tables which map them. This structure implies that segments start on page boundaries. As a result of sharing second level page tables, no second level page table should be used in mapping more that one segment.

Sharing page tables makes no difference to address translation but must be taken into account when manipulating page tables. The fact that each segment is mapped to a file affects the strategy used to find where the page must be read from. When the address cannot be translated because of the absence of a page table entry for the virtual address indicating the page is not in memory or an operation caused a violation of permissions an exception called a page fault is generated by the memory management hardware. A page request is illegal if it is for an unpaged part of the kernel, an unmapped part of the user address space, or if it violates the permissions on a segment. No paging is done on the kernel address space so a page fault for any kernel address indicates an operating system error.

When a page fault occurs the page fault routine determines the identity of the missing page. The segment map is used to find which page within the segment is missing. The shared file entry pointed to by the segment map is used to determine which file is to be read and the block corresponding to the missing page. The process sleeps while the block is being read. The use of existing kernel file access routines during paging ensures the use of the readahead mechanism if pages of a segment are being used sequentially. When the read is finished, the process which was sleeping on paging is woken up and an entry created for the page in the second level page table.

It is possible, when a page fault occurs, that the second level page table does not exist. Allocating a second level page table is not completely straightforward because processes with the same file mapped into their address spaces share second level page tables. The shared file array can be used to determine if the second level page table exists elsewhere in the system. Associated with each shared file array entry is a list of second level page tables being used to map blocks from the file which are in memory. When allocating a second level page table this list is searched to see if the table already exists so an entry for the table can be placed in the process's level 1 page table. If it does not exist a new page table is allocated and an entry added to the list.

4.4 *Freeing Level 2 Page Tables*

Periodically all the pages belonging to a process are examined to see if they are still in use; if not, they can be released. Pages are also released when a segment is removed. When no pages are left in a second level page table it also can be released, but the release algorithm must take into account the ability of page tables be shared by some other process.

To release a second level page table the entry in the process's first level table is invalidated. The reference count for the table in the shared file array is decremented and the memory used by the page table physically released if the reference count is zero.

4.5 *Setting Up Segments*

The system call *makeseg* creates a new segment in the address space associated with a named file. This involves creating a segment map entry, opening the file and creating a shared file entry or associating the segment with an existing one. The system call, if successful, returns the virtual address of the base of the segment. *Makeseg* will have the following parameters:

1. The name of the file to be mapped in.

2. The virtual address, on a page boundary, of the start of the segment. A suitable base address will be chosen if this is not specified.

3. The number of pages contained in the segment which will default to the number of blocks in the file if this parameter is zero.

4. The offset in blocks within the file where the mapping is to begin.

5. Possible flags are: permissions for the segment, to prevent the segment from being deleted by the user particularly the text, data and stack segments created by *fork* or *exec*, allowing exclusive access to the file making the segment non-sharable, requesting that the file be created if it does not exist, and to have the file truncated on opening.

If the file exists the user must have the appropriate permissions on the file. A file that does not exist can only be created if the user has appropriate permissions on the directory. The existing file system software is used to validate the file name and open the file. The fourth parameter allows a part of a file to be mapped or even different parts of a file to be mapped to different segments such as when data and stack segments are both mapped to the same process image file. Segments are prevented from overlapping within the virtual space. The size of a segment can be larger than the file if the segment and the file both have write permission. In this case the file can be extended.

4.6 *Extending Segments*

The system call *exseg* will be provided to enable a segment to be explicitly extended upwards or downwards. *Exseg* replaces the *brk* system call which increases the amount of real memory allocated to a process. The parameters are:

1. The virtual address at which the segment starts, which is used to identify the segment.
2. The number of blocks (or bytes rounded up to the nearest block) by which the segment is to be extended. If this is positive the segment is extended upwards, and if negative the segment is extended downwards.

The call fails either if there is no existing segment with the specified base, if extending would cause the segment to overlap another segment or overlap the begining of the file it is mapped to. Extending beyond the end of a read only file is a non-productive activity which should be prevented. The virtual address of the segment base is returned if the call is successful. If the segment was extended downwards the base address may have changed.

4.7 *Removing Segments*

The system call *rmseg* will remove a segment mapping. It is passed the virtual address of the segment base and removes the corresponding entry from the segment map array. If the segment has write permission any modified pages are written to the file before closing it. The reference count in the shared file map entry is decremented but only deleted when the number of references falls to zero.

4.8 *Getting Information about Segments*

The system call *segstat* can be used to obtain information about the segment whose base address is passed as a parameter. The information is returned into a buffer specified by the second parameter.

4.9 *Fork, Exec and Exit*

In the same way as the child inherits the open files of its parent when forked it shares all the sharable segments in its parent's address space. The process image file and non-sharable segments are duplicated for the child. Since an *exec* overlays the calling process with the contents of the named file, all segments including files mapped to segments by *makeseg* are released. When a process terminates *exit* removes all segments and deletes the process image file.

4.10 *Design Problems*

A problem arises when a page has been modified but is not in a buffer because it is still in use resulting in an unmodified copy of the information being read from the file when another process reads the file. This may happen if a file is being read or written using a combination of access methods, say a process has it mapped into its address space and another process is reading it in the usual manner. The simple solution is to only allow one type of access at a time. Another solution is to force modified pages to be written every time the process is switched out where the file is being shared.

5. *Conclusion*

Extensions to the UNIX Version 7 kernel have been described which provide the functionality of System V's *regions* and of Version 8's */proc* directory but is more uniform in approach than either of these. The extensions involve fairly localized changes to the kernel after alterations to provide paged virtual memory have been made.

6. *References*

[Bach 86] Bach, Maurice J. *The Design of the UNIX Operating System.* Prentice-hall Inc., New Jersey, 1986

[Killian 84] Killian, T. J. *Processes as files* European UNIX Systems User Group, Autumn Meeting September 1984

Recent Work on Research Unix

Peter J. Weinberger
UNIX Research Group
Bell Laoratories, AT&T, USA
research!pjw

Work continues on modifications of version 9 of research Unix, in networking, in file system types, and in coping with our new VAX 8550. I shall survey the present and immediate future with an unseemly emphasis on my work on network file systems.

THE SHARED LIBRARY MINEFIELD

Michael Selig

Olivetti Australia Pty. Ltd.

ABSTRACT

You have just installed Unix* System V Release 3, and have read all the glossies about the benefits of Shared Libraries, yet there are some niggling questions in the back of your mind.

- Can I really make use of them?

- What overheads are involved when using them?

- What benefits am I really likely to get?

- Are they easy to use?

- How much work will be involved to convert my programs to use them?

This paper attempts to shed some light on the use of Shared Libraries and help to answer these questions.

1. HOW THEY WORK

The traditional Library under the Unix Operating System is a concatenation of a group of object modules (".o" files) in a single file. The format of this file is such that the Linker (ld) is able to scan the Library to resolve references from the user's program to external objects (both functions and data items). When such a reference is resolved, the object module from the Library is appended to the executable file ("a.out" file). Once the external object is assigned an address, all references to that object can be fixed to refer to this address. The Linker only picks out those modules from the Library that are actually needed, so that the amount of memory that the final executable program requires is as small as possible.

Using this Library mechanism, several programs, all using the same routine, will each have their own copy of that routine (Fig. 1). When Shared Libraries are used, however, the code for the subroutine is not appended to the executable file, but is kept in the "Target Library" file. This is shared by all programs using the Library (Fig. 2). As a result, using Shared Libraries generally means that executable files are smaller, and when several programs using the same routines from a Shared Library are executed concurrently, less memory is required.

The code contained in the Shared Library is mapped at runtime into a fixed address within the address space of each program that uses the Library. This mapping works in a similar way to Shared Memory Segments but is done automatically when the program is executed by the exec(2) system call. As a program may use several Shared Libraries, each must have a unique base address so that they do not clash, and all must be mapped to addresses that cannot clash with the non-shared code. On the 3B2 computer, for example, the virtual addresses 0x80000000 to 0x80800000 are reserved for Shared Libraries.

If a Shared Library is modified, the addresses of the routines are likely to change. Therefore all references to routines in the Library are done via a Jump Table. As long as the positions in the Jump Table are not changed, routines can be altered, and new ones added without having to relink the programs that use the Library.

* Unix is a registered trademark of AT&T in the USA and other countries

2. HOW TO USE THEM

When a program is linked with a Shared Library the Linker uses a file called the Host Library. This file is in the normal library archive format that the Linker uses, but instead of containing the real routines, it consists of 'stubs' which define the addresses of the routines and data items in the Target Library.

To use a Shared Library rather than a normal one, all that is required is to link the program with the Host Library file.

For example, if the non-Shared C Library is called 'libc.a' and the Host Library file of the Shared C Library is called 'libc_s.a' then the command:

```
ld prog1.o libc.a       will use the non-Shared Library, and

ld prog1.o libc_s.a     will use the Shared Library.
```

Thus the use of Shared Libraries is very easy, and re-compilation of the programs that use the Library is not necessary.

When a program requiring a Shared Library is executed, the Operating System automatically maps the Text and Data regions from the Target Library file into the process' address space. This operation happens transparently, so that running programs which use Shared Libraries is no different to any other program.

3. BENEFITS AND DRAWBACKS

Up to this point, the Data regions of the Library have not been discussed. In the current implementation, if a process uses a Shared Library, all Text (Instruction) and Data regions in the Target Library are mapped into the process' address space whether they are needed or not. Since the Data regions cannot be shared, this can cause a process' Data region to be larger than expected.

Furthermore, in this implementation, all external references from within a Shared Library must be satisfied every time a program is linked with the Library. Thus, if a routine in the Library references an object in a non-Shared Library, this object will be linked into every program that uses the Shared Library, even if the routine that references it is not used by the program.

However, if care is used when the Shared Library is built, these situations can be avoided.

3.1 DISK SPACE

As explained in the previous section, when a program is linked with the Host Library file, the routines are not copied into the a.out file, but are stored in one file, the Target Library. Therefore with the exception of the situation described above, using Shared Libraries always results in smaller a.out files.

3.2 MEMORY

As long as several programs using the same Shared Library are likely to be executed concurrently there will virtually always be a saving in memory requirements. However, in a demand paged environment, the amount of memory that a process needs is not as important as in a swapped system; it is more important to reduce the number of page faults (see below). Again, the situations described at the beginning of this section can actually lead to increased memory requirements, but this is unusual.

3.3 SHARED LIBRARY CAN BE MODIFIED WITHOUT RELINKING PROGRAMS

Since the code from the Shared Library is stored in one file only (the Target Library) rather than storing copies in each program, when the library is modified, all programs using it will immediately use the new code.

Because the routines in the Library are called via a jump table, whose order is not changed, they can grow and shrink without the user having to worry about their addresses changing.

3.4 PAGE FAULTS

As the code in a Shared Library is shared by several concurrently executing programs, it is likely that the commonly used routines will stay in memory. This means that when a new program starts and calls one of these routines, there will not be a page fault, whereas a program using conventional libraries would probably cause one. On the other hand, rarely called routines in a Shared Library will almost certainly cause a page fault when called because they unlikely to be present in memory. If such a routine is linked into the program in the normal manner, there is less chance of a page fault because it may already be in memory due to its proximity to other routines in the same program.

3.5 OVERHEADS

There is a very small overhead using Shared Libraries, as the routines are called indirectly. As discussed in section 4, objects outside the Library must be referenced indirectly from routines within the Library, adding another insignifiscant overhead.

Benchmarks have been run comparing demand paged systems using the standard C Library versus the Shared C Library, which show that total system performance is very similar[2]. A lot more work is yet to be done to compare the performance of other more specialised Libraries, such as database or graphics Libraries. J. Arnold [2] feels that in a more controlled environment that usually exists in a real world computer system, the use of say a shared database library, should decrease paging activity and hence increase system performance.

In summary, the benefits of using Shared Libraries clearly outweigh any drawbacks as long as the Library has been built with the possible pitfalls in mind.

4. HOW TO CREATE THEM

When a Shared Library is created, all external references from within the Library must be satisfied. If this were not the case, all references would have to be linked at runtime, causing a considerable overhead. This leads to the problem: How does a routine in a Shared Library call a routine outside that Library whose address may be different from program to program (Fig. 4a) ? The solution used is to reference all objects outside the Library indirectly through a static pointer (Fig. 4b). This pointer is initialised to the address of the object by each program. Such a reference is known as an 'Imported Symbol'.

Unfortunately, this means that changes must usually be made to the source code of a Library to make it Shared. It also means that it is almost impossible to convert an existing standard library to a Shared Library without access to the source code.

The work to change the source code can be made less painful by using the C Preprocessor. For instance, if the routine 'printf' is to be imported, an include file, say "import.h" can be created containing the line:

```
# define printf (*sl_printf)
```

This file must be included in each source file that uses 'printf'. As well, the pointer must be declared by adding a line to a file called say, "import.c":

```
int (*sl_printf)() = 0;
```

These files "import.h" and "import.c" should have a declaration for each imported symbol. Of course, "import.c" must be compiled and included as a module in the Shared Library.

To make matters even more tricky, it is not always easy to find all the symbols that must be imported. This is especially true if the Shared Library uses the Standard I/O Library which implements some routines as macros. The best way that I have found to determine what symbols need to be imported is to try to build the library without importing any, and then to note all the undefined symbols that are listed.

In order to build a Shared Library, a Library Description file should be created. A sample file appears below:

```
#target      /shlib/libtest_s
#address     .text      0x80680000
#address     .data      0x806a0000
#branch
             proc1      1
             proc2      2
#objects
             proc1.o
             proc2.o
#init   proc1.o
             sl_printf   printf
             sl_strcpy   strcpy
#init   proc2.o
             sl_printf   printf
             sl_read     read
             sl_write    write
```

This file contains:

- The base address of the Text & Data regions of the Library.

- A list of the routines in the Library. This list is used to build the Jump (or Branch) Table into the Library and, as mentioned before, the order of this table (and hence this list) must not be changed once the Library has been used.

- A list of the object files to link together to form the Library.

- A list of the imported symbols and their values.

This file is used as input to the "mkshlib" command, which then builds the Host and Target Libraries.

The process of converting existing Library source code into a form suitable for a Shared Library is very tedious and some simple tools to make this process easier should be provided. It would not be hard for instance, to write a program to automate the production of the 'import' files described above.

Another short-coming of the current implementation is that there seems no way of avoiding the importing of a symbol that is known to be in another Shared Library, and hence has a fixed address. For instance, if our Library calls 'printf', it would be nice if we could force it to call 'printf' from the Shared C Library, because then the address of the routine would be known, and we would not have to import it. Unfortunately, this cannot be done.

If it were possible to avoid having to import any symbols, then no source changes would be necessary, and you could then convert a normal Library to a Shared Library without re-compiling. In practice, this is virtually impossible as nearly every Library at some stage, calls a routine from the C Library.

As you can see, building a Shared Library is not a trivial task, and cannot be done without access to the source code of the Library.

5. SUMMARY

The implementation of Shared Libraries in Unix System V Rel. 3 is very flexible, and once they are created, are very easy to use. Any normal user can create and use Shared Libraries, as they do not have to be configured into the operating system as is required in some implementations.

Furthermore, as long as a Shared Library has been created properly, it will result in savings in disk space and memory, with improvements in system performance, and no appreciable overhead. Building the Libraries, as we have seen, is not as easy as may first be thought. Also, to make the library perform as efficiently as possible, a lot of work must be done profiling programs in an effort to reduce page faults. Only after this has been done, is it possible to determine whether a particular routine should be shared or not. Also, extra tools to aid in the construction of Shared Libraries should be provided which could make their building as easy as that of a conventional Library archive.

6. APPENDIX

GOLDEN RULES OF SHARED LIBRARIES

1. Do not include small, infrequently called routines.

2. Keep static and global data to a minimum. Prefer to use 'malloc' to allocate buffers etc, rather than declaring them as data.

3. Try to keep references to objects external to the Library to a minimum. In particular, avoid using external data items - prefer to use a pointer passed as a parameter.

4. Try not to export data items at all. Prefer to pass data back to the calling program via a function return value or through a pointer passed as a parameter.

5. Preserve the order of the Jump Table.

References

1. Unix System V Release 3 Programmer's Guide, AT&T Technologies.

2. Shared Libraries on Unix System V, James Q. Arnold, $echo, AT&T Unix Pacific Co. Ltd., Feb 1987.

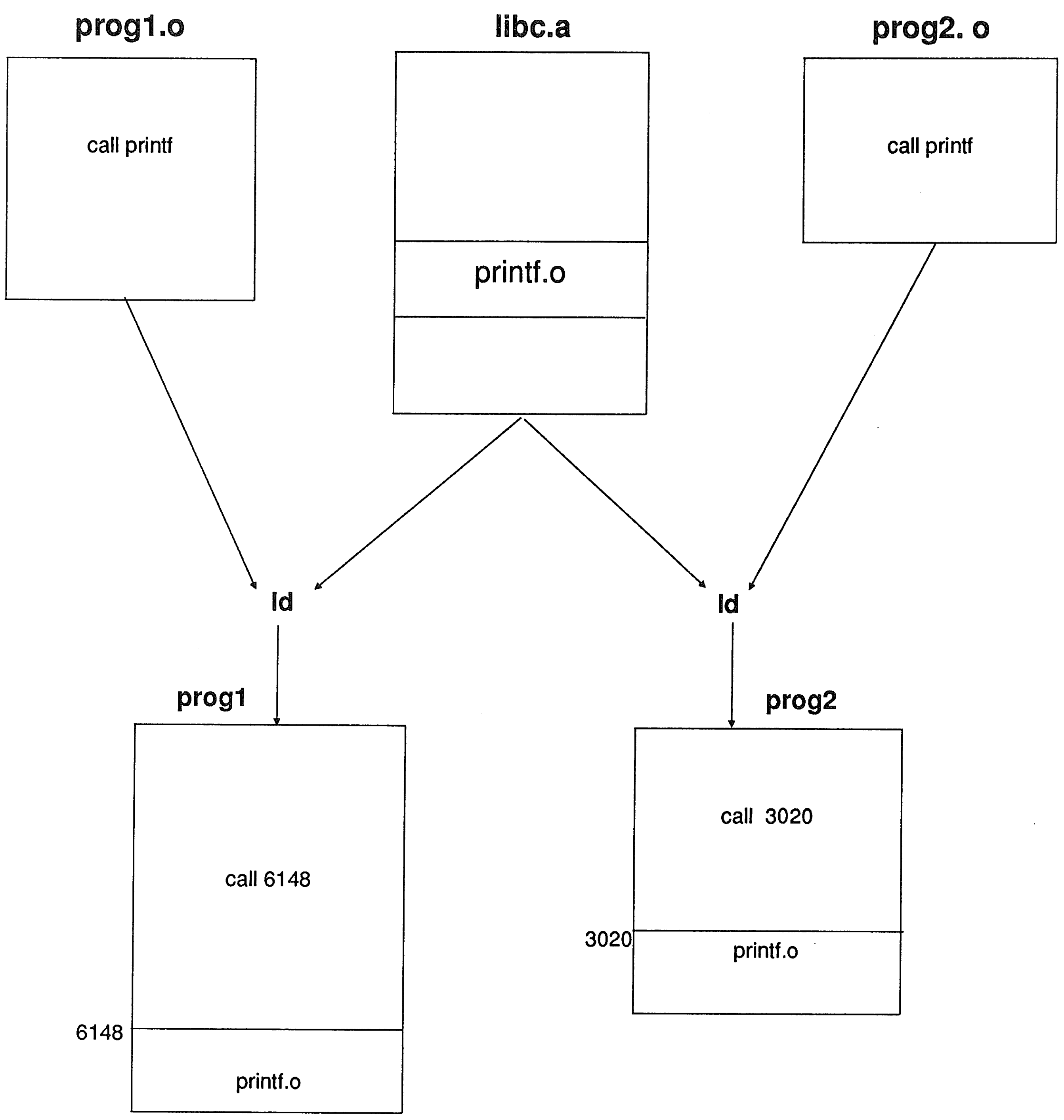


Fig. 1 Use of Conventional Libraries

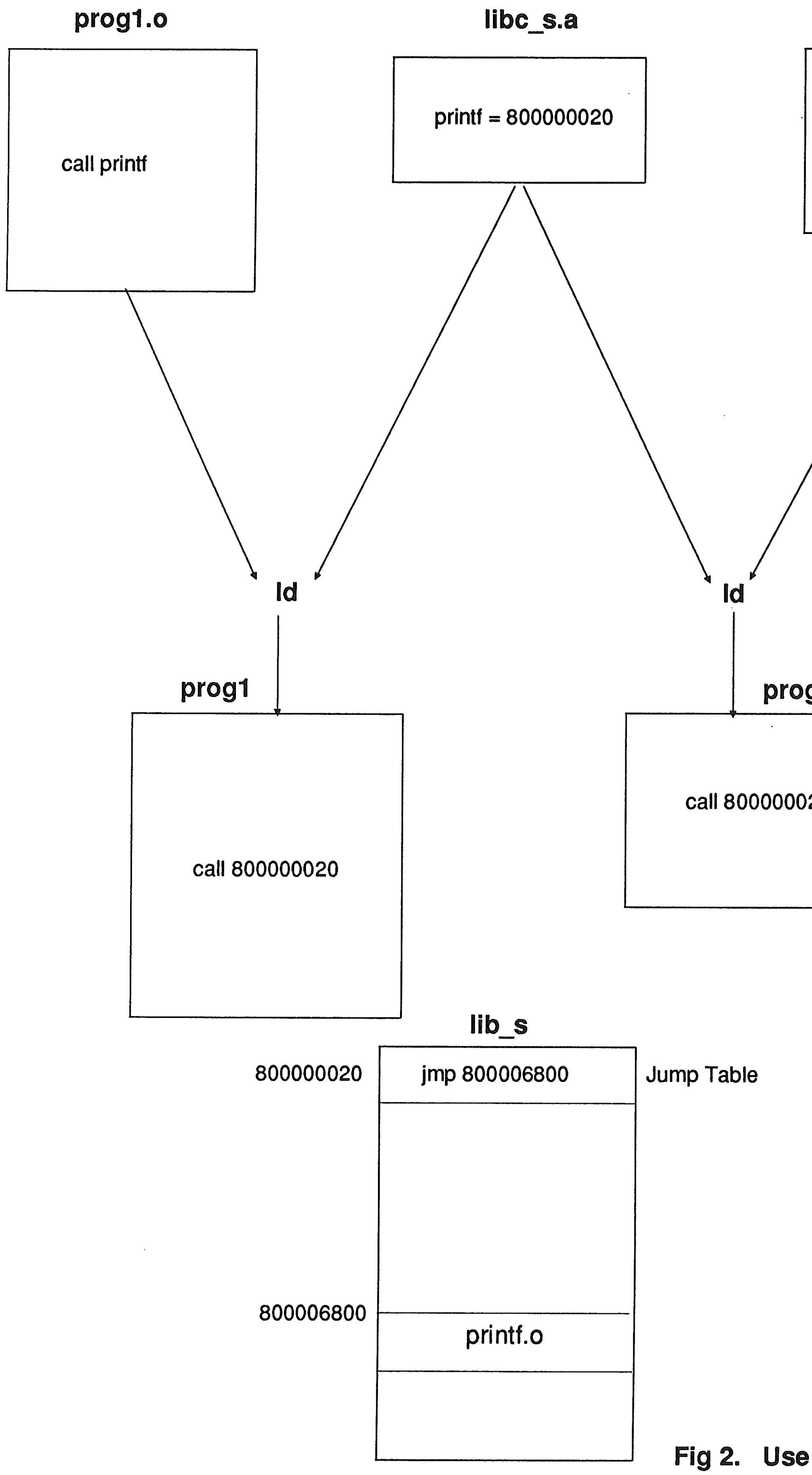


Fig 2. Use of Shared Libraries

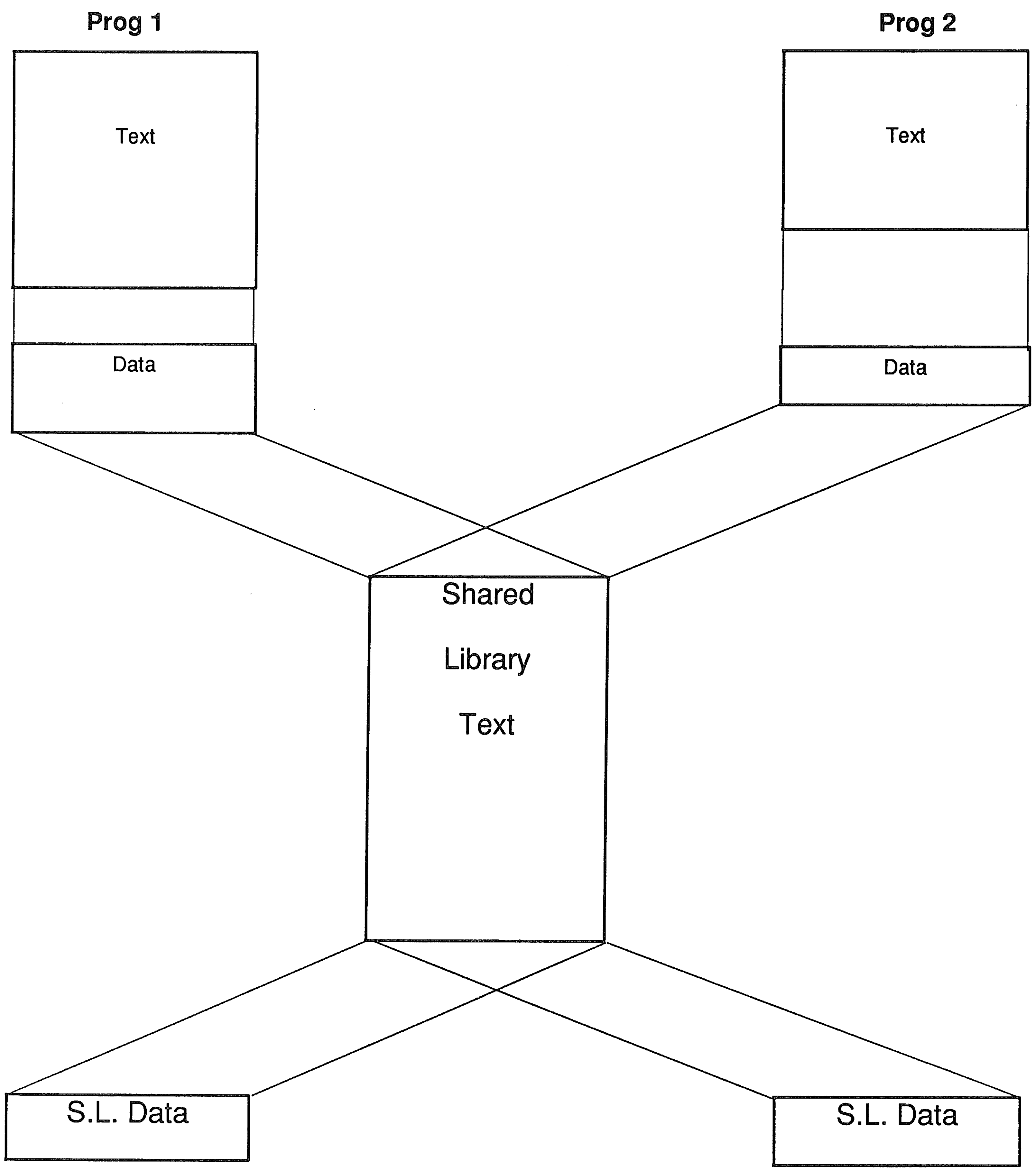


Fig. 3 The Shared Library is mapped into the address space of each program that uses it.

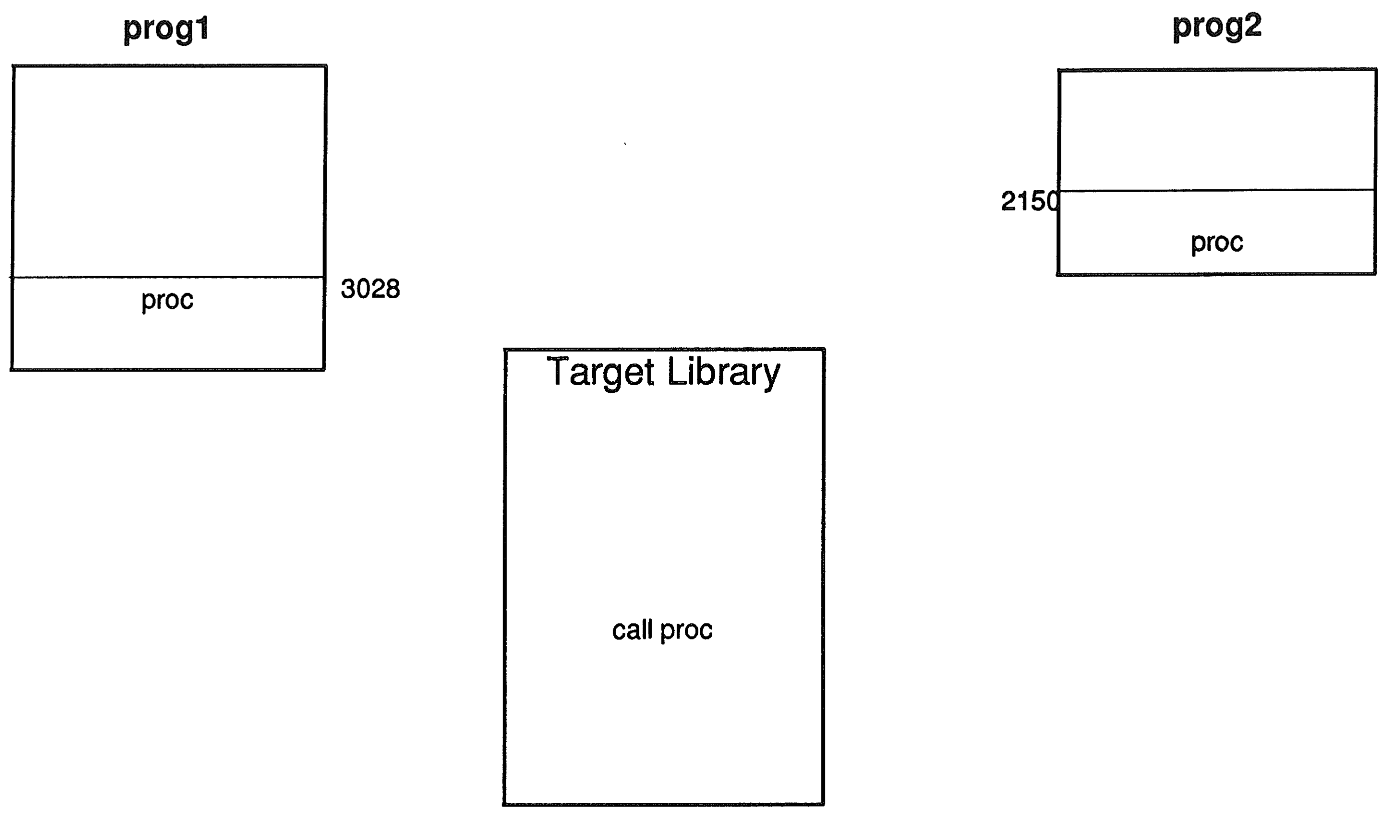


Fig. 4a A dilema: The address of 'proc' is different in the 2 programs that use the Shared Library.

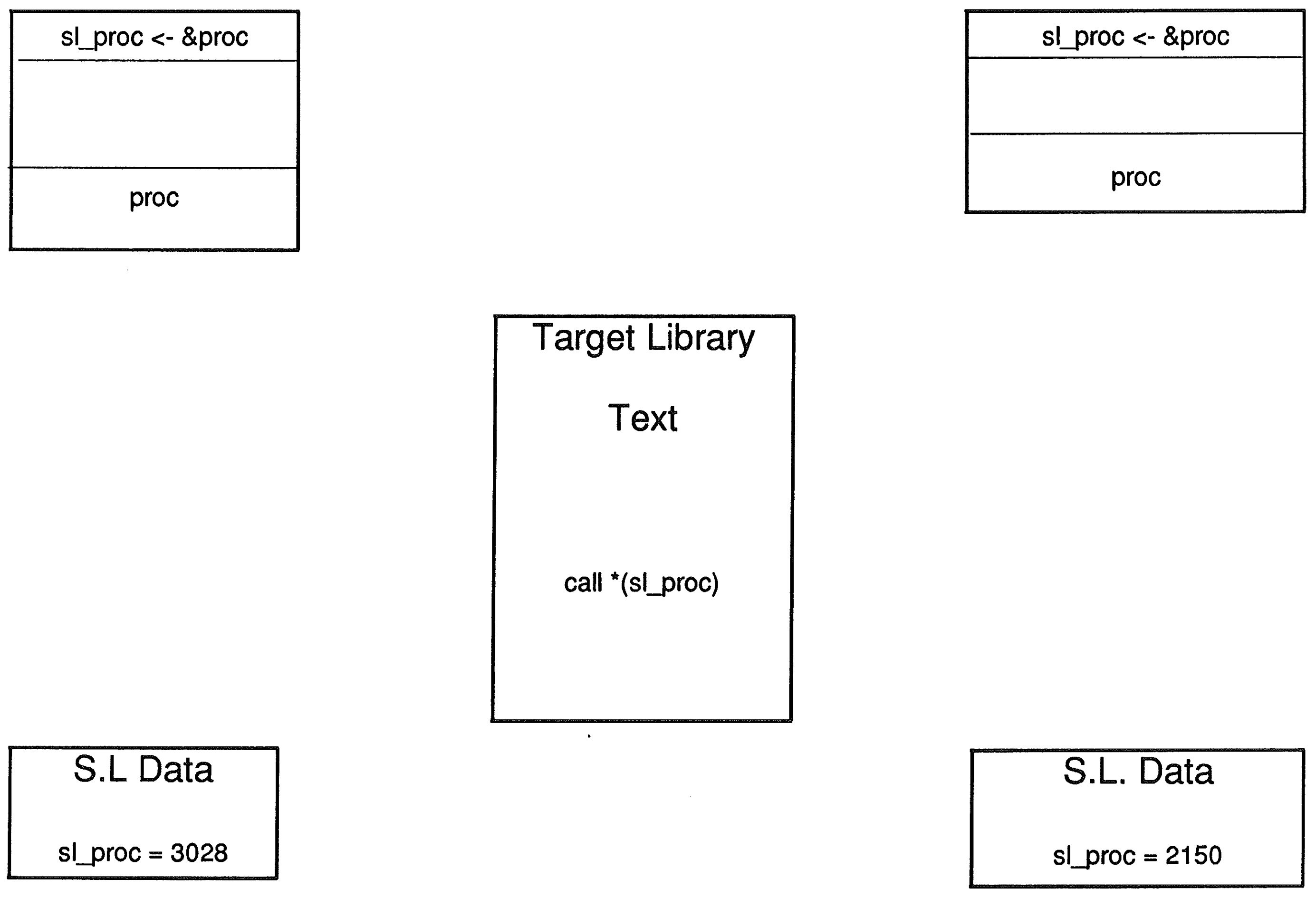


Fig. 4b The solution: Call the routine indirectly, and Import the value of the routine when the program starts.

SunOS Release 4.0

Rich Burridge

Sun Microsystems Australia,
123 Camberwell Rd,
Hawthorn, VIC 3122.
richb@sunk.oz

ABSTRACT

SunOS Release 4.0 is targeted for availability in the first quarter of calendar year 1988. This paper summarises the planned content. It is being issued early in the recognition of our customers' needs for advance information to accommodate their product planning efforts. All contents and schedules are subject to change without notice. Sun will, to the best of its ability publicise any such changes.

A. Positioning Overview

SunOS Release 4.0 follows the Sun tradition of:

- o promoting a single UNIX standard converging System V and BSD features,
- o providing maximum functionality,
- o incorporating the latest state-of-the-art operating system technology, and
- o supporting computing in heterogeneous environments

This release of SunOS includes many enhancements for the installed base:

- o extended converged UNIX functionality,
- o addressing key systems administration needs, and
- o creating a foundation for the anticipated hardware architectures

The Sun UNIX kernel has been significantly redesigned as another step in the on-going process of incorporating state-of-the-art technology to establish a stable architectural base for the next decade. This base includes work to incorporate the latest virtual memory management techniques, shared libraries, and improved system security

While this type of implementation work is essential to maintain the leading performance and functionality of SunOS, Sun is committed to retaining industry standard interface compatibility. In this manner, implementation changes can be accomodated while maximising code portability to this and future SunOS releases.

B. Contents Overview.

The following is a partial list of the new features in 4.0:

Support for Sun-4

Release 4.0 supports the new Sun-4 product family workstations. This involved changes to the kernel, as well as to the compilers.

New Architectural Features

Features:	*Description or comments:*
1. Major New Kernel Architecture	The kernel has been significantly restructured to accomodate a new virtual memory management scheme that promotes sharing system resources. Swap space requirements are reduced, system resources are utilised more efficiently, cacheing of frequently accessed data is more efficient, and files can be treated as part of virtual memory making access to large files more convenient and efficient. The new VM system provides sharing on a page-by-page basis and employs a copy-on-write mechanism to create individual copies of pages as they become necessary. Swap area is now a regular file sharing disk resources with other files.
2 Shared Library Facility	Using shared libraries reduces program size and swap space requirements and provides the flexibility to incorporate new versions of libraries automatically, thus facilitating bringing new libraries into system-wide use. *libc* and user built libraries are sharable in this release and future releases will offer more system libraries converted to shareable format. Shared libraries utilise the new virtual memory system and a revised link editor (*ld*). The C compiler has been enhanced to generate position independent code (pic) and *ld* supports either linking code dynamically at run-time or statically at initial program load. Note: Sharing and its associated dynamic binding is the current planned default operation although a small performance penalty may result from run-time linking.
3. Resizable swap area for diskless clients	Resizing client swap space no longer requires taking a server and all its clients off-line or reinstalling the system. Only the client(s) being modified needs to be suspended.
4. Lightweight Processes	Provides the ability for a C program to have multiple concurrent threads of execution. Lightweight processes facilitate developing programs that manage asynchronous events. Facilities provided include messages, monitors, exception handling, and flexible context switching. Both coroutines and preemptive scheduling are possible.

4.0 provides lightweight processes only at the user application level. Kernel support is not provided.

5. Other Kernel Enhancements

These include:

a. System V shared memory implementation now supports paging of shared data space.

b. Network interface tap (NIT) improvements include packet filtering and integration with the streams environment.
Packet filtering provides an Etherfind-like capability allowing applications to monitor ethernet traffic selecting, for example, ethernet packets from a certain host or with a specific field setting.

Networking

1. Improved support for diskless clients

Diskless clients are supported via the NFS file system. *nd* and its associated partitions no longer exist. This improves heterogeneity support (non-Sun servers can now support diskless Sun clients) and improves administration since swap area is now a regular file. (Note: Being a regular file also means that client swap areas shares disk resources with other applications running on the client and with other clients.)

The performance goal is to achieve Release 3.2 performance levels in the absence of *nd*. Eliminating *nd* does NOT impact the number of clients a server can handle.

A boot block server will be provided to support Sun-2s with proms that don't support tftp booting, so no prom change is required by this change.

2. Filesystem re-org

The re-organized filesystem facilitates maintenance of servers by permitting more sharing of binaries, matching diskfull and diskless file layouts, easier mixing of multiple architectures, and easier access to remote executables in general. A single server can more easily support multiple architecture clients and non-Sun servers can support diskless Suns.

The *root* filesystem and */usr/lib* have been re-organised.
Shareable files, including architecture dependent files, are now all in */usr/lib* and client private data is all in *root*.

4. YP Enhancements

The YP databases can be updated over the network. This simplifies system administration and, depending on local administrative policy, the system administrator need no longer be solely responsible for incorporating YP map updates.

5. Secure networking features

Secure networking improvements provide more thorough authentication of user identification prior to allowing file access. Security is provided by supporting the exchange of conversation keys and by preventing superusers from masquerading as other users to access secure filesystems across the network.

NFS utilizes secure RPC so that servers can optionally become more secure. Secure operation is selected by specifying the *-secure* option for individual */etc/exports* entries. For secure operation RPC utilises a YP database of public and private conversation keys.

6.	NFS performance	NFS benefits significantly from the improved cacheing improvements provided by the new virtual memory system.

Standards

1.	System V enhancements	New features include:
	a.	Remaining system calls required for Base Level Release 2 functionality: *chown, creat, fcntl, kill, mknod, open,* and *utime.*
	b.	Complete System V *STREAMS* interface incorporated in the kernel. Using STREAMS simplifies writing device drivers by supporting portable protocol modules.
c.		Fully System V and BSD compatible tty interface using *STREAMS.*
	d.	System V compatible archive utility (*ar*).
	e.	System V batch utility and job scheduler (*at, batch, cron and crontab*).
	f.	Access to Sun added-value libraries (e.g., SunView) from System V programs.
2.	More 4.3BSD	Most 4.3BSD functionality is provided, including the new upper limit of 64 (vs. 30) on the number of open files per process and a fully 4.3BSD compatible subnet facility.

In general, 4.3BSD provided many bug fixes and performance enhancements. Most of those which are applicable to Sun Workstations and which were not already in SunOS prior to the availability of 4.3BSD or added since then are provided by SunOS Release 4.0. (Note: Sun Consulting has developed an implementation of XNS that includes the Transport and Network layers on top of standard ethernet.)

Peripherals

1.	New *setup* utility	The new *setup* utility provides convenience and flexibility. New capabilities include the ability to reuse configuration files for common configurations, to edit existing configuration files to correct errors or support minor configuration differences, and for sophisticated users to specify variants to Sun supplied configuration alternatives.

setup runs off a table interface much like the 3.x terminal version of *setup*; there is no window version.

2.	On-line disk formatting utility	The disk formatting utility now runs under UNIX and can be used to format any drive not currently mounted. Formatting disks is now much faster and no longer requires a dedicated system. Multiple disks can be formatted simultaneously.
3.	New Mass Storage Systems	Support for as yet to be announced mass-storage systems is provided.
4.	Removal of Interphase 2180 driver	The driver for the Interphase 2180 Disk Controller is no longer included in the SunOS. The removal of obsolete drivers is part of the on-going program to unburden the standard system software from supporting an ever growing number of devices. As drivers become obsolete, the pending removal at the next release is announced. Systems (a few Sun-2/100Us and Sun-2/150Us) with Interphase 2180 Controllers must be upgraded before they can run Release 4.0.

Other Enhancements

1.	Secure System Work	Optional security measures provide an auditing log of failed and/or successful attempts to use system resources on both a system wide or individual user basis and identifying who was using the system at the time of attempted security breaches. Improved protection of the password database is also provided. In DoD parlance, Release 4.0 provides "unevaluated C2" level security. This is the groundwork for a future secure system product.
2.	C Global Optimizer	Significant run-time performance for C routines is now possible via the C compiler optimization option which applies the same global optimizing technology that is available in Sun's FORTRAN compiler.
3.	Internationalization	8-bit integrity will be assured at this release in order to support non-ascii character sets. This is the first step towards an internationalized UNIX that supports local character sets. The 8-bit kernel and tty driver support creating file names with 8-bit characters from within C programs. However, utilities such as *suntools, csh, sh, vi* and other tools do not incorporate this 8-bit filenaming at this release.
4.	Documentation	New manuals, and improved introductory guides and system administration guides ease the tasks of mastering and administering the system. New documents include: *System Services Overview* and a Global Index to all system manuals. Documents with major revisions include: *Installing UNIX and Programming Utilities.* Repackaging of the documentation offers the convenience of selecting the set of manuals targeted to the specific users needs, e.g. introduction to system use, system reference, system administration, and program development.

5.	Obsolescence Mechanism	Instigation of */usr/old* directory and movement of obsolete modules to that directory. This introduces a warning system to prepare users that the included modules will be removed in the next release. For Release 4.0, the following programs will be moved to */usr/old*:

sun3cvt	Was only needed for 2.0 to 3.0 transition.
compact	Replaced by faster and more efficient but incompatible *compress* program from 4.3BSD.
eyacc	Was used only to implement Pascal. Has been removed in 4.3BSD.
make	Pre 3.4 version of *make*.
prmail	Replaced with *mail -u* by 4.3BSD.
pti	Replaced with *troff -a* by 4.3BSD

C. Release Specifications

Performance

Release 4.0 is expected to achieve performance levels at least equivalent to that of Release 3.2.

Resource Requirements

Release 4.0 resource requirements (memory and disk) for the system software are expected to be the same as for Release 3.2. Shared libraries have contributed to making the basic resource requirements of Release 4.0 comparable to those of 3.2 in the presence of added capability. The full set of system software and associated files (including text files such as man pages and font files which do not benefit from the space savings provided by shared libraries) will likely require more disk space than consumed by Release 3.x files.

Code Migration and System Administration Changes

Maximizing binary compatibility is a major design objective of Release 4.0. To date no known widespread sources of binary incompatibility for well-written code exist. As incompatibilities arise they will be carefully documented. Source code changes may be required in some binary compatible programs when they are recompiled. The following lists key known compatibility issues and areas subject to visible impact on administrative or program-build procedures:

Features:		*Compatibility Issues & Procedural Changes:*
New Architectural Features		
1.	Major New Kernel Architecture	Programs that read the structure or depend on the format of core files may require revisions.
2.	Shared Library Facility	Build procedures for libraries may need to change. Existing programs will work but will require rebuilding to take advantage of shared libraries. Build procedures for programs requiring a non-shared library must explicitly select this option.
3.	Resizable swap area	New, easier administrative procedures required.
4.	NIT Improvements	Programs utilizing NIT will require source changes and recompilation.

Networking

1.	Improved support for diskless clients	New, easier administrative procedures required
2.	Filesystem re-org	New, easier administrative procedures required.
3.	YP Enhancements	Potential changes to administrative procedures.
4.	Secure networking	Pre-4.0 (insecure) yppasswd may not work against a secure 4.0. yp yp server. Secure authentication will prohibit access to setuid programs lacking authorised access.

Standards

1.	System V enhancements:	
	SVID Compliance	Programs using *creat, fcntl, kill, mknod, open,* and *utime* may have to be relinked.
	System V/BSD tty	Programs using System V *termio ioctl* calls will perform better if they are relinked.
	Batch utility/job scheduler	Slight impact on system administration procedures only.
2.	System V shared memory	Programs using the System V shared memory *shmdt* library call that were linked on a Release earlier than Release 3.4. should be relinked to avoid a spurious message to the console window.

Peripherals

1.	New setup utility	New, easier administrative procedures required.
2.	On-line disk formatting utility	New, easier administrative procedures required.
3.	Removal of Interphase 2180 driver	Systems using Interphase 2180 must be upgraded.

Other Enhancements

1.	Secure System Work	Modules using the encrypted password field in */etc/passwd* or */etc/group* will require modification.

Installation

Release 4.0 is a major release introducing extensive architectural changes. As such, it requires a full system installation. With the new setup utility, system installation and configuration is, however, much more convenient.

D. Acknowledgements.

This paper is mainly based on an internal pre-release report written by Pat Harding, plus a presentation given to Sun personnel at Palm Springs in late July.

E. References

[GING 87} Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, "Shared Libraries in SunOS", Summer Conference Proceedings, Phoenix 1987, USENIX Association, 1987.

[GING 87] Gingell, R. A., J. P. Moran, W. A. Shannon, "Virtual Memory Architecture in SunOS", Summer Conference Proceedings, Phoenix 1987, USENIX Association, 1987.

[CHEN 87] Cheng, R., "Virtual Cache in UNIX", Summer Conference Proceedings, Phoenix 1987, USENIX Association, 1987.

Writing Parallel Programs for the Sequent Multiprocessor

Stephen Frede
Softway Pty Ltd.

The Sequent family of computers are symmetric multiprocessors. In order to take fullest advantage of the multiple processors available, a single application has access to specialised parallel programming primitives. This talk is a discussion of those primitives in the context of the C programming language and the Dynix (parallel UNIX clone) operating system.

Unix on the Cray

Peter J. Weinberger

Bell Laboratories has had a Cray XMP-24 for about 18 months. During most of that time the machine ran both COS and Unicos, Cray's version of Unix. Recently it started running Unix on both processors. Supercomputers are unlike the usual run of Unix machines, and I shall talk about our experience, and implications for the system.

A low cost, short range, reconfigurable microwave data link *

Chris Clarkson, Ian Dall & Alex Dickinson

1 Introduction

Many institutions, such as research facilities and universities, that utilize large scale computer services are distributed geographically, with buildings being separated by distances of up to several kilometers. Historically such institutions have had computer services provided by a centralized facility with user terminals connected via serial lines. The more recent availability of minicomputers, workstations and powerful personal computers has led to a change in the distribution of computer facilities. Processors are in general moving closer to the user, often onto the desk. This redistribution has in turn increased the need for computer interconnect. A great deal of a computers power as a tool comes from its information *sharing* capability. Users can share programs, data and messages and this move to distributed processing requires networking facilities that can maintain these services as users move off central machines.

This paper describes a flexible networking link based around digital and microwave hardware. The link has been designed to support a variety of different networking needs in a single, user reconfigurable unit:

1. *Distributed Processing.* Interconnection of mainframes, minicomputers, workstations and file servers is supported by the provision of fast TCP/IP capable links, allowing use of higher level facilities such as NFS and ACSnet.

2. *Mixed Processing.* Interconnection of personal computers is supported by the provision of simple links that appear as serial lines thus economically interconnecting personal computers via serial ports and the use of simple communication facilities such as Kermit.

3. *Centralized Processing.* Interconnection between ordinary terminals, printers or personal computers to central machines is also provided by the provision of simple links that appear as reconfigurable serial lines.

*Copyright ©1987 C. Clarkson, I. Dall, A. Dickinson

The unit itself has a number of advantages in addition to its flexibility in terms of networking tasks:

1. *Channel reconfigurability.* The unit may be configured with between four and one hundred and twenty-eight channels by installation of the required number of four channel sub-systems. The baud rate of each channel may be separately selected, with the channels being multiplexed onto a single 500k bits per second link.

2. *Interface simplicity.* The unit provides a link that is transparent to the user. RS-232 connections, which can make use of either XON/XOFF or DTR/CTS flow control, are provided at either end and appear to be connected by a "virtual wire" provided by the link hardware and software. However the microwave medium is inherently less reliable than a "physical wire". Therefore the protocols within the link ensure that it appears to the user as a one hundred percent reliable "virtual wire".

3. *Installation simplicity.* The only requirement on transceiver installation is line-of-sight capability. This has obvious cost and time advantages over installing cabling, without cabling's poor expansion capacity.

4. *Recurring costs.* The recurring costs are independent of volume of data transfer, basically equipment servicing and a yearly licence fee. Telecom's proposed move to time based local digital data charging makes this style of link attractive as an alternative to local modem links.

In this paper the system design, communication protocols and hardware implementation of the link are described.

2 System Design

The system design of the link is partitioned at the uppermost level into digital and microwave modules. The microwave module is regarded as providing a single, high bandwidth, *unreliable* serial link. The creation of multiple, *reliable* link channels becomes the province of the digital module, and it is the design of this digital module that is described in this section.

The digital system is partitioned into three modules, each providing an increasingly abstract layer of communication services. This structure is analogous to that provided by the lower levels of the International Standards Organization Open Systems Interconnection model. Interactions between layers occur only through well defined interfaces that operate on a client-server basis, the lower levels acting as servers (clients) to the higher levels for incoming (outgoing) data.

The three modules are:

1. *Input Level:* The Multichannel Processor (MCP).
2. *Intermediate Level:* The Polling Processor (PP).
3. *Link Level:* The Link Management Processor (LMP).

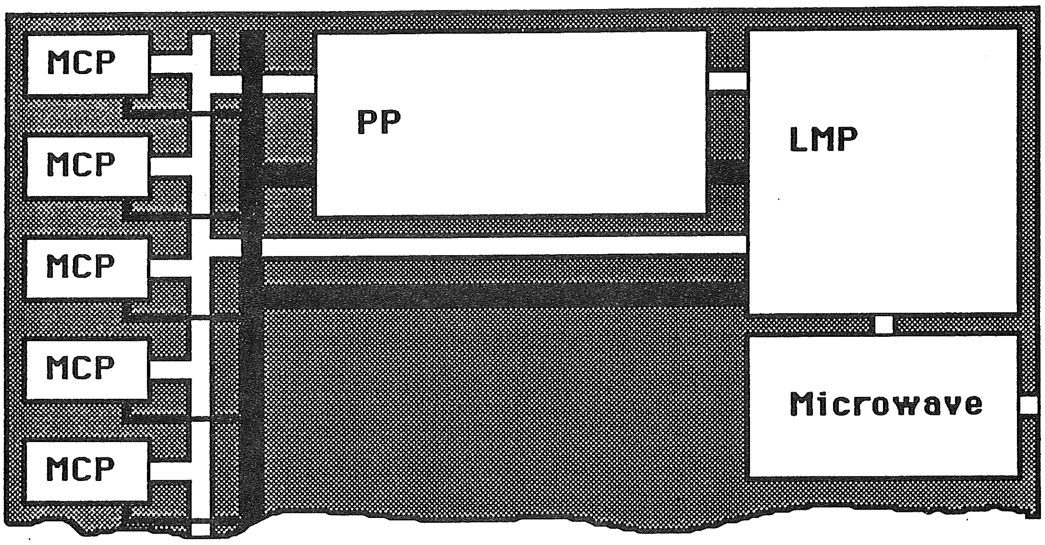


Figure 1: System overview.

The overall system design is illustrated in Figure 1. The operation of the three digital modules is described in the following subsections.

2.1 The Multichannel Processor

The input level section provides the interface between the system and the RS-232 data circuits. It has been named the Multichannel Processor (MCP). The MCP provides for an initial level of data concentration/distribution. An MCP supports four RS-232 ports. The configuration of speed, flow control interpretation and parity is facilitated for all the four channels. The design of the MCP is shown in Figure 2.

The MCP operates in the following manner. Whenever a byte of incoming data is received on any one of the four channels the MCP reads this data and attaches an address to it. This address is derived from which channel supplied the data and where the MCP is in the system backplane. The MCP then writes this data into a queue which is presented for the next level to read. The MCP must in addition read from a second queue, which another layer writes outgoing data into, and transfer this to the appropriate RS-232 circuit.

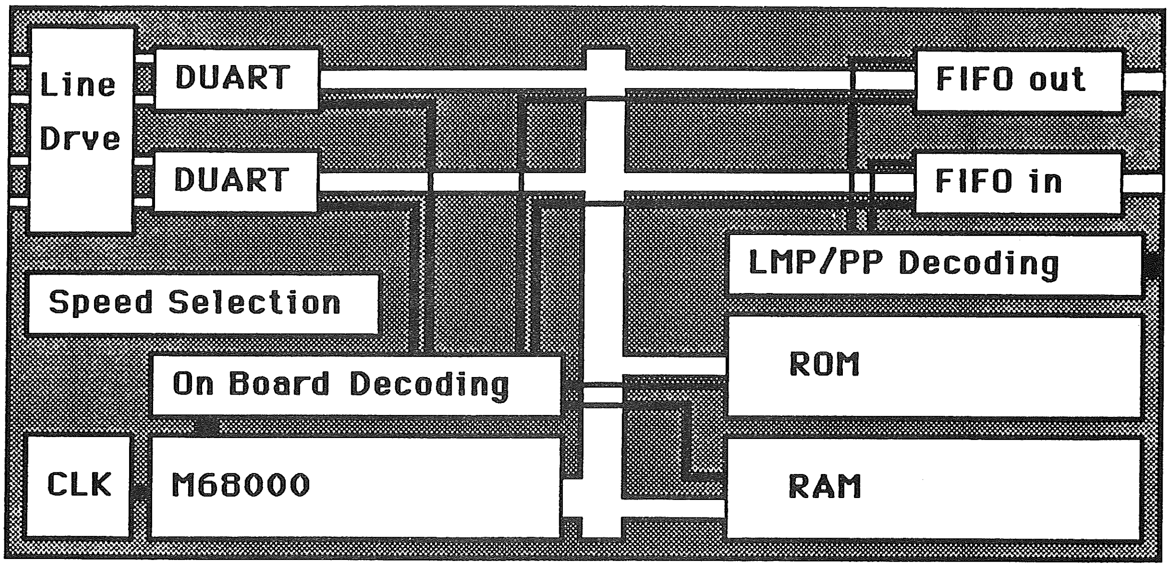


Figure 2: Multichannel Processor (MCP) overview.

The MCP must also cope with flow control of the channels. Flow control can either take the form of in band XON/XOFF or out of band DTR/CTS. An additional flow control consideration for the MCP is the filling of its internal queues and the prevention of loss of data. The MCP accomplishes these tasks by means of in band signaling of both out of band data and control information by means of special packets. The MCP must therefore keep track of the state of the flow control on each channel. Options are provided for interpreting XON/XOFF or not and utilizing DTR/CTS or not.

2.2 The Polling Processor

The Polling Processor (PP) is responsible for polling the MCPs and presenting the collected data to the next level. At this level a statistical multiplexing scheme could be implemented. However in the prototype design a straight poll of the MCPs is carried out. The complexity of this section could be decreased by reducing this processor to nothing more than a multiplexer, a counter and some combinational logic. However using a microprocessor provides a high level of flexibility in implementing efficient polling strategies. The design of the PP is illustrated in Figure 3.

2.3 The Link Management Processor

The Link Management Processor (LMP) is responsible for the majority of complex high level link functions. The LMP must read data from the PP and assemble it into packets to be sent to the microwave module. It transfers these packets to devices that subsequently feed the microwave hardware. It must hold copies of these packets in memory

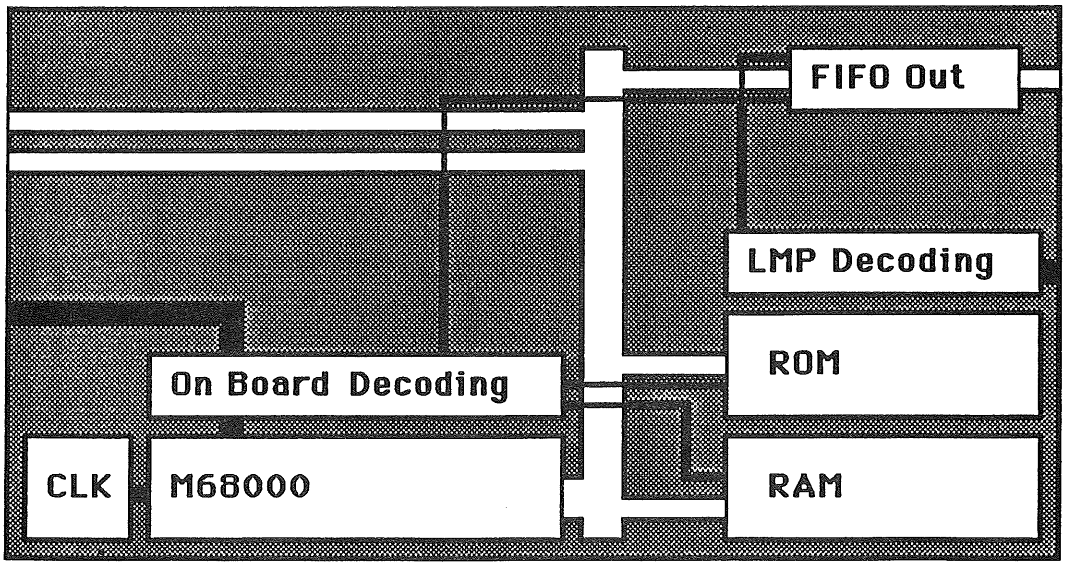


Figure 3: Polling Processor (PP) overview.

until it has received positive acknowledgement of their receipt. If positive acknowledgement is not received for a packet then the LMP must resend it. Therefore these packets must contain packet identification, acknowledgement or otherwise of sent packets, and error checking information. The LMP must also disassemble incoming packets, checking their validity, placing acknowledgement information in outgoing packets and finally present the data to an MCP. The design of the LMP is illustrated in Figure 4.

2.4 Performance

The link data rate is limited by two factors, the raw bit rate of the microwave system and overheads within the digital system. Radio frequency bandwidth limitations dictate a raw bit rate of 500k bits per second between the microwave components. There are three sources of overhead in the digital system. The first comes from constructing packets in the MCP to be passed to the next level. The second source of overhead comes in the additional information required in the LMP packet constructs. This is in fact minimal and only of the order of five percent because of the LMP packet size. The final overhead is introduced by packet storage and retransmission. This overhead is somewhat indeterminate but may be as high as twenty five percent depending on the protocol used. From these consideration a figure for the effective bit rate of the link may be derived and the baud rate of a channel when the maximum number of MCPs are present. It can be shown that the effective bit rate of the link is approximately 200k bits per second, implying that the average baud rate of a channel when all MCPs are equally loaded is a little more than 1200 baud. However much higher peak baud rates on each channel can be achieved.

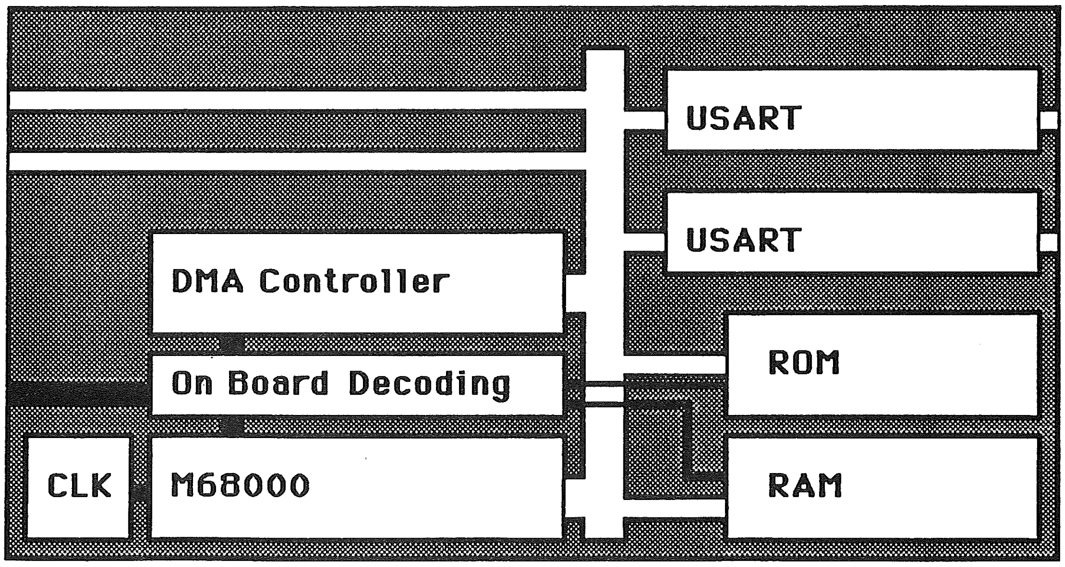


Figure 4: Link Management Processor (LMP) overview.

3 Communication Protocols

The system essentially utilizes only two protocols. One of these is concerned with the concentration and flow control handling within the MCP and is internal to the sending and receiving systems. The other protocol appears at link level and is used to obtain reliable end to end transfer of data over the unreliable microwave media. This is the more complicated of the protocols.

The MCP packet protocol entails a concentration step and signaling for out of band data and control information. The MCP packet has three elements. The first element is the data that is being sent. The second element is the address of the channel the data is destined for. The final element is a flag signaling that the packet is either a conventional packet or a packet that contains out of band or control information.

The link level protocol is an error detection and retransmission protocol as opposed to an error detection and correction one. A sixteen bit Cyclic Redundancy Check is used to perform packet verification. A windowing system is therefore implemented to facilitate the resending of packets out of order. The link packet has four elements. The first element is packet identification information. The second is acknowledgement information used for packet verification. The third element is link status information. The final element is a stream of MCP packets followed by CRC information. The protocol is tailored for use on a synchronous link.

4 Hardware Design

The system is packaged in two units, the microwave transceiver hardware and the digital hardware. The two are connected by a single high speed RS-422 serial data link. This enables the microwave unit to be placed in a position with line-of-sight access to the other end of the link and the digital unit to be placed in a position such as a machine room.

It should be noted that the digital hardware is independent of the medium employed and could be applied for use with other unreliable physical links such as line of sight infra-red or laser links.

4.1 Digital Hardware Unit

The design of the digital hardware is a compromise between performance, flexibility and cost. Microprocessors are used in all three modules of the digital system because of their low cost and flexibility. However, in several cases novel designs for interfaces between microprocessors and the rest of the associated hardware have minimized the processing to be done in software. This occurs with little or no extra hardware costs and with minimal reduction in flexibility. Extensive use has been made of hardware FIFO queues in communicating between the digital system components. This allows a very simple interface between the three sections without the complexity of multi-ported memory which would otherwise be necessary to achieve sufficient performance.

The Motorola 68000 processor has been selected as the processor for use throughout the system. While in some cases an eight bit processor such as the Zilog Z80 may have been adequate, the cost difference would have only a minimal effect on the overall cost of the system, and the extra processing power would prove invaluable if more sophisticated algorithms are used in future developments.

Each MCP consists of a 68000 processor, two dual asynchronous receiver transmitters, an input FIFO, an output FIFO plus associated RAM, ROM and a small amount of SSI "glue". The output FIFO of the MCP connects with some control logic to the PP via the split system bus (SSB). The PP always controls one half of the SSB and so avoids all bus latency and arbitration overheads.

The PP consists of a 68000 processor, RAM, ROM an output FIFO and some SSI "glue". The "glue" logic has been kept to a bare minimum by using a very simple address decoding scheme. This is possible because the physical address space of the 68000 is much larger than will be used in this application. The PP communicates with the LMP via a FIFO.

The LMP consists of a 68000, RAM, ROM, a four channel DMA controller, two Universal Synchronous Receiver Transmitters with in built CRC generation and testing and associated SSI "glue". Most of the data handling is done by the four channel DMA controller which transfers data from the input FIFO to RAM, from RAM to the output USART, from the input USART to RAM and from RAM to the SSB.

Data is transferred from the LMP to the MCP via the other half of the SSB. The LMP simply places the MCP packets on the SSB. No extra processing is necessary since the packets contain the address of the destination MCP.

4.2 Microwave Hardware Unit

The system utilizes a simple microwave transceiver capable of full duplex operation. The transceiver operates in the X band of the Ultra High Frequency ranges. The transmitter incorporates an amplitude modulated ten milliwatt Gunn diode oscillator. Superhetrodyne technology is employed within the receiver. Modulation and demodulation are accomplished using standard amplitude modulation techniques. An RS-422 channel is provided for communication with the digital hardware unit. The design of microwave hardware is illustrated in Figure 5.

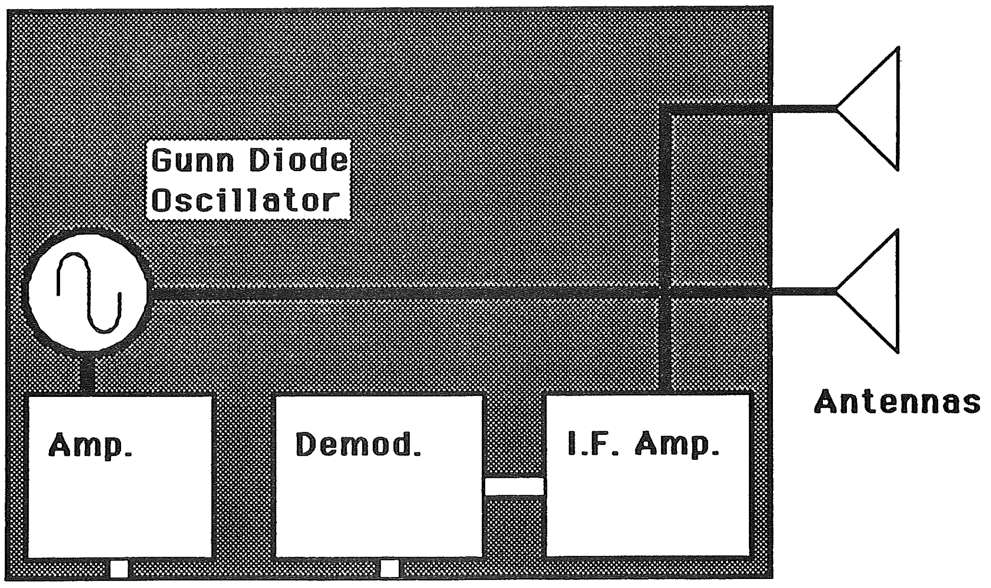


Figure 5: Microwave hardware overview.

5 Conclusion

This paper has described a versatile, reconfigurable and extendable microwave data link for use in economically providing a spectrum of networking services to localized sites.

A prototype link is currently being produced. It is to be installed between the University of Adelaide computer centre and a university residential college. The link will provide for the college's growing needs by virtue of its extendability, as more ports are required, further MCP boards may be installed.

Acknowledgements

Many thanks to the other members of the design team: Nick Davias, Paul Franzon, Mike Pope and Mike Liebelt.

Some Aspects of System V.3 Networking

Tim Roper
timr@labtam.oz

Labtam Limited

ABSTRACT

Several features new to Release 3 of System V are intended to support the implementation of network protocols and applications. STREAMS provides the underlying mechanism for implementing network functions as layered kernel modules with well defined interfaces. AT&T has tendered its Transport Interface (based on the ISO Transport Service Definition) as the common interface to be presented by transport providers to higher layers of kernel modules and user programs. This framework is supposed to set the scene for the future development of network protocols and services in System V.

The STREAMS mechanism and the Transport Interface will be described. Remote File Sharing and cu/uucp will be cited as examples of applications using them. Cursory comparisons with alternative systems may be made.

1. Introduction

The UNIX† operating system has frequently been used as a vehicle for research into and experimentation with new ideas in computer networking. However the traditional character I/O system was designed with slow terminal lines in mind. It handled data a character at a time with minimal processing concerned only with the user interface (such as input editing).[1] For high bandwidth communication technologies and complex network protocols it lacked efficiency and generality. Various enhancements were made to the system over time. Efficiency improvements included handling groups of characters together where possible. Some functionality was added with *line disciplines.* Of course *ad hoc* changes did not improve the general appeal (structure, maintainability, extensibility) of an area of the system that had long been complicated. Dennis Ritchie redesigned it. His *Stream Input-Output System* was described, appropriately, in the October 1984 special issue of the Bell Laboratories Technical Journal.[2]

When seeking to redress the lack of support for networking services in UNIX System V, AT&T chose Ritchie's Stream I/O System as the basis for the new framework on which such services could be implemented. They called their variant STREAMS.[3]

By predefining the *service interfaces* of certain levels of protocols AT&T intends System V to help the *implementation* of both applications that are independent of the protocols whose services they use, and of higher level protocols that are independent of the implementation details of lower level ones. In particular a *Transport Interface* has been defined.

To allow user processes to access services provided by STREAMS modules in the kernel various enhancements were made to the user/kernel interface. The lack of a good method for user level programs under System V to do I/O on several devices in a demand driver manner (''synchronous I/O multiplexing'') has been dealt with to a limited extent.

These new facilities are available in Release 3 and are introduced in this paper. It should be stated at the outset that some other UNIX operating systems have features which address similar problems as

† UNIX is a trademark of Bell Laboratories.

these. A brief reference to them is made.

2. STREAMS

2.1. Motivation

Figure 1 shows the protocol components of a file transfer session using DARPA Internet protocols over Ethernet.[4,5] Data flows from the server process to the client process via the protocol modules on the server's host, the physical medium and the protocol modules on the client's host, and *vice versa*. It undergoes processing in the various modules mostly concerned with managing the communication as well as the file transfer. We are concerned with using STREAMS to implement the flow and processing of the data within a host running System V Release 3.

2.2. The Structure of a Stream

Figure 2 shows a simple use of STREAMS. A device *driver* in the kernel is connected by a full-duplex connection, called a *Stream*, to a user process. It is created when the user process opens a character special file that is distinguished in the kernel configuration as identifying a STREAMS device rather than a traditional character device. The Stream is then referenced with the returned file descriptor as with traditional character special files. All Streams start out this way. A Stream disappears from the system when last closed.

The Stream *head* manages the user/kernel interface (Section 3). That is, it implements those system calls relevant to STREAMS. For example, it copies data between a user process's memory and a Stream in response to system calls.

A more interesting Stream may be constructed by *pushing* a *module* onto it. This causes the module named to be interposed in the Stream at the top and hence to receive, presumably process, and re-send messages flowing up or down the Stream. Conversely the module currently at the top may be *popped* off. Each time a module is pushed a new instance of it is created. Modules, drivers and heads communicate by passing pointers to messages; copying is generally avoided. Modules, like drivers, are implemented in the kernel; a given module must have been configured to be pushed.

Suppose that the device in Figure 2 is a serial interface (presumably with a terminal attached). Data written to the Stream is transmitted to the terminal with no post processing (eg. tab expansion). Input read by the process has not been subject to *canonical processing* (eg. no carriage-return to line-feed mapping). A request for parameters from the Stream provides the baud rate and number of stop bits being used but not the current erase or interrupt character. In Figure 3 a module implementing traditional *tty* features has been pushed onto the Stream. Data put onto the Stream by the process and the driver will now be processed in the manner of traditional ttys.

Alternative line disciplines may be implemented as distinct modules. Note that the hardware dependent details have been contained in the driver. The generic, hardware independent, terminal functions have been placed in a module that may be used above any serial device driver. In fact it may be pushed onto any Stream that is associated with a terminal user, such as a network connection to a remotely logged in user.

2.3. The Structure of a Message

The unit of data on a Stream is a message. Whether a module or driver respects the boundaries between successive messages depends on the semantics that it implements. The serial interface driver in Figure 3 would ignore message boundaries in both directions treating each character as the unit of data. The *tty* module may assemble incoming characters into lines and place each line in a separate message so that the process reads one line at a time as is expected of ttys. However, boundaries between messages from the process are likely to be arbitrary, perhaps corresponding to the process's output buffer size. A module doing packet assembly/disassembly may treat data to and from the disassembled (start-stop) side one character at a time whereas each message to and from the assembled (packet mode) side may correspond to one packet on the network. Any meaning attached to message boundaries is by convention between cooperating modules, drivers and processes. Of course structure information may be encoded in the data using other conventions.

Since modules, drivers and processes communicate by passing messages a mechanism is required for them to pass control information. This is done by tagging each message with a *type*. The set of message types is fixed and includes the following.

DATA	user data, protocol headers etc.
PROTO	service primitives and their parameters (Section 4)
IOCTL	an encapsulated user process *ioctl* call
IOCACK	
IONAK	the response from the driver to an IOCTL
ERROR	a driver notifying the head of an error condition
STOP	
START	
DELAY	a request to a driver to start, stop or delay transmission

Some message types flow in sequence with DATA and are subject to flow control. Other types are given priority, jump to the head of queues and are never held up by flow control. For example PROTO messages are in the first category. This is necessary as they may delimit state changes in the data stream. A PROTO message that enables or disables a data encryption module would cause havoc if it travelled out of sequence with the DATA messages concerned. On the other hand a STOP message is given priority as it is assumed that the remote end of the communication has requested, in a protocol dependent manner, that transmission be stopped immediately. For example, in Figure 3 it may be the *tty* module that recognises XON/XOFF characters from the remote end and signals the driver with START/STOP messages.

Memory for messages is allocated in fragments of a small number of fixed sizes (''quick fit''). A message consists of one or more chained fragments of the same or different types. For example an IOCTL fragment is followed by a DATA block containing the arguments to the call.

2.4. The Structure of a Module

A module is composed of two halves. The *write* half handles data flowing downstream from user to device. The *read* half handles data flowing upstream from device to user. For each instance of a module there are two *queue* structures, one for each half, called the *read* and *write* queues. These queues may be used for buffering data, if required by a module. Messages are queued by linking the head of the first message to the queue and linking the head of each subsequent message to the head of the previous message. In any case queues serve as the kernel's handle on the module (halves).

Data arriving at a module from upstream may be immediately processed and passed onto the next module downstream. Or it may be put on the module's write queue pending later processing. Similarly, data arriving from downstream may or may not linger on the read queue. The entry points to a module consist of two functions for each direction, the read and write *put* and *service* routines. Put routines are mandatory; service routines are optional. When a message arrives at a module the appropriate put routine is called with (pointers to) the appropriate queue and the message as arguments. Passing the queue distinguishes between multiple instances of a module. At the choice of the module designer, the message may be immediately processed and passed onto the next module. Or it may be placed on the queue. In that case the associated service routine is implicitly scheduled to be called at some later time (if the queue was previously empty).

Service routines are called with a queue as their sole argument. By writing modules so that they save state in a ''per Stream'' data structure they can be made to function a bit like coroutines. One of the queue structure's fields is an otherwise unused ''generic'' pointer that is usually initialised by the module (when it is pushed onto the Stream) to point to that Stream's data. Hence the queue argument establishes the module's state for the particular Stream. It is important to note that modules are not processes, but just kernel subroutines. The only context supplied them by the kernel is the queue argument.

Service routines are called by the kernel scheduler and may be interrupted. Put routines are generally assumed to be called at (hardware) interrupt time and hence with interrupts masked. The module designer can therefore separate processing into a priority part that must be done as soon as possible even if interrupts are disabled, and a less urgent part that may be done with interrupts enabled.

2.5. Multiplexing

Although a module may be pushed onto many Streams each instance is independent. A driver, however, may be involved in many Streams simultaneously. The serial interface driver in Figure 4 is handling a device with several ports. It must be concerned with aspects of the system's single interface to the device as well as the device's many interfaces to its attached lines. Such a driver is said to have *upper multiplexing*. Individual Streams are created by opening different *minor* devices.

A hardware device driver is a special case of the need for multiplexing when implementing networks. More generally, multiplexors may be required at intermediate points in a Stream. Implementing a transport protocol that provides many process to process connections on one host to host link requires upper multiplexing (Figure 5). Alternatively, an internetworking protocol deals with messages from many network interfaces, hiding higher level protocols from this fact (Figure 6). This is called *lower multiplexing*.

STREAMS provides multiplexing by generalising the notion of a driver. Multiplexing drivers have a minor device for each Stream that they are willing to connect to simultaneously. Streams above a multiplexing driver are created by opening the device as with device drivers. Attaching a Stream below a multiplexor is more complicated. The head of the Stream that is to be multiplexed is connected to the lower side of the multiplexing driver. This is done by opening the device and issuing an *ioctl* call on it with the lower Stream as an argument. The lower Stream is then accessible only through the multiplexing driver until the process explicitly breaks the connection with another *ioctl* or closes the device. A lower multiplexing driver is configured by requesting many such connections below it. A driver may be both upper and lower multiplexing. For example, the ARPAnet Internet Protocol[6] is designed to connect to multiple networks and to dispatch packets to different higher level protocols depending on a field in the IP header (Figure 7). Note that Stream configurations are determined at run time from a set of components determined when the kernel is configured.

2.6. Flow Control on a Stream

The STREAMS flow control mechanism is tied to queues and service procedures. Before passing a (low priority) message on, a module is expected to check that there is room for it in the next module's queue. If there is not, it is expected to keep the message on its own queue. When the next module's queue has enough room, the first module's service routine is automatically called to restart the data flow. In this way, congestion at a driver causes data to back up in the previous module, then the one previous to that, and so on. This will extend back to the Stream head, if necessary, causing any further writes by the process to block and put it to sleep. When the driver's output queue drops sufficiently the previous module is re-enabled and its queue then drops, re-enabling the module previous to it. Eventually the Stream head is re-enabled, the process wakes up and the write call completes. A module (or more correctly a read or write half of a module) that has no service routine takes no part in flow control.

If modules obey the rule about service routines checking for space before passing a message to the next module, they need not be concerned with flow control. The system takes care of suspending and re-enabling modules. This is because one instance of a module takes part in only one Stream. However multiplexing drivers are more complex. They take part in multiple Streams. In general, data entering on one Stream may leave on any other. The automatic re-enabling of service routines across modules does not happen for drivers. Modules have exactly two queues, a read queue and a write queue. Each queue has a put and a service routine. But drivers have a pair of queues for every upper and lower connection point. It is up to the driver to route messages between them according to whatever protocol it is implementing. Drivers can have different upper and lower read and write put and service routines, a total of eight entry points (as well as those for initialisation). In the example of Figure 7, congestion in the serial driver should not stall the IP driver and inadvertently affect the Ethernet. There are separate queues on the lower side of the IP driver for them. It is up to the driver writer to decide how congestion on the lower side should be reflected on the upper side, and *vice versa*. For example, a multiplexing protocol that flow controls virtual circuits individually could base advertised window sizes on the state of the appropriate upper Stream. Note the distinction between Streams flow control and network flow control.

3. The User/Kernel Interface

As mentioned in the previous section, the Stream head translates between user process system calls and messages on the Stream. This includes suspending and rewaking a process according to the requirements of flow control. In addition it can affect a process by posting a signal on it at the request of a module or driver, such as when a terminal user types the interrupt character.

There are several new system calls in System V.3 associated with Streams. The most obvious are those for writing data to and reading data from a Stream, *putmsg* and *getmsg*. Unlike *read* and *write*, each call specifies two buffers using a *control* and a *data* parameter. Hence module control information is separated from ordinary data. The control parameter of a *putmsg* call specifies data to go in a PROTO fragment. The data parameter specifies data to go in as many DATA fragments as required. The message placed on the Stream consists of the DATA fragments linked behind the PROTO fragment. Conversely, any PROTO fragment in a message arriving at the head is transferred to the process according to the control parameter of a *getmsg* call and the data fragments are transferred according to the data parameter. Note that the PROTO part is not intended to contain a protocol header but rather a request to or an indication from a service provider.

A common need when writing user level programs providing network services is to be able to do I/O on many (real or pseudo) devices in a demand driven manner. An example is a window manager that multiplexes data for many pseudo-terminals onto one serial line. The problem is that initiating a read on one pseudo-terminal when no data is available causes the process to sleep awaiting its arrival, oblivious to data output by other pseudo-terminals or input from the line (ie. reads block). A similar problem exists when writing, as a device whose output queue is full may cause the process to sleep until it drains (ie. writes block). Previous releases of System V have allowed the programmer to specify immediate return from a I/O calls that cannot be immediately satisfied. The problem is often not that the program doesn't want to block, it just doesn't want to block on one device when it could be doing I/O on another. STREAMS avoids these problems with the *poll* system call. A process can determine which of many devices can be read from or written to immediately. It may elect to carry on if none can be read immediately, or to block until at least one can be and/or a timer expires. Unfortunately, *poll* is only applicable to Streams. Under System V.3 the pseudo-terminals mentioned above could be implemented as Streams devices and/or modules but if the real device is not driven by a Streams driver we still have a problem.

There are some situations in which it is desirable to expedite the processing of data to or from a Stream over other "background" processing. In this case the programmer can request (per Stream) that a signal is to be posted when an "interesting" condition has occured, such as the arrival of data. System V signal handling has been improved to make this feasible.

4. Service Interface Definitions

Applications that require only simple, common network services could ideally be written in a network independent manner. That is, the same program could be run over different underlying networks simply by giving the name of the service provider as an argument. A similar ideal is that where a higher level protocol requires only a common, simple service from a lower level protocol, it should be possible to use implementations of the two protocols of diverse origin with no re-writing.

STREAMS provides a general framework for implementing protocol modules. It does not enforce the compatibility of programs with protocol modules or the compatibility of implementations of protocols from the same family (let alone different families). In addition to STREAMS, in System V.3 AT&T have promoted the idea of protocol independent applications via standard service interface definitions.

4.1. Transport Interface

AT&T have defined an interface to be presented to higher levels by transport service providers. Also, they have developed some guidelines for achieving protocol independence in applications. Known as TLI, the Transport Interface is modelled on the ISO Transport Service Definition.[7] Note the difference between a service definition and a service interface definition. The former defines the services to be provided in an abstract manner, for example the primitives. The latter defines how they may be accessed, for example how requests for service are encapsulated as subroutine calls or messages. Standards bodies usually deal with the former definitions and deliberately avoid the latter.

A kernel level service interface is via messages passed on Streams. The messages are structured as a PROTO fragment whose data contains a code for the primitive required plus any parameters, followed by zero for more DATA fragments containing user data. Such messages may be read and written by a user process with the *getmsg* and *putmsg* system calls. For the Transport Interface, a library is provided to simplify programming. It includes routines corresponding to each service primitive.

Future service interface definitions for System V are expected. For example a well accepted link level service interface definition could allow one vendor's Ethernet controller (with driver) to be used with another's Internet Protocol implementation without modification of source code by any party.

4.2. Uucp

Uucp is a package that provides store and forward transfer of files and jobs between machines running UNIX operating systems.[8] Historically this has used dedicated serial lines or connections dialled over a telephone network. Where a more sophisticated network is available it may still be useful to provide *uucp* services in the interests of uniformity.

With System V.3 the configuration management of *uucp* has been modified. One modification is the addition of a routine to set up calls over any network that implements a TLI compatible transport service. A host on such a network can be contacted by *uucp* by configuring the transport provider as the device, the host's transport address as the telephone number, and the device type as TLI rather than one of the other known types (such as diallers).

Of course *cu* is similarly enhanced. These are examples of protocol independent applications implemented as user level programs using TLI.

4.3. Remote File Sharing

An example of a transport service user implemented in the kernel is AT&T's Remote File Sharing.[9] This is a mechanism for extending the file system of one machine with those of other machines on a common network, thus making remote files transparently accessible. (Its functionality may be compared with the Network File System[10] from Sun Microsystems.) When RFS is started (usually at boot time) the kernel is informed of the TLI compatible transport service to be used. This service is also used by various RFS support processes such as name servers. It is possible to shutdown RFS and restart it using a different transport service without rebooting the machines.

5. Protocol Migration

It is interesting to consider whether STREAMS lends itself to protocol *migration*. By this we mean moving the implementation of a protocol between user and kernel or between kernel and hardware (or firmware.) It is usually more convenient to debug user level code than kernel code. But a user level implementation may not perform as well as a kernel implementation. Debugging an implementation as a user level program and then putting it into the kernel may therefore be attractive. (Of course if the rewriting required to fit it in is too great this approach may not be worthwhile.) Sometimes it may help to devise Stream configurations that provide the required inter-process communication. For example, a pseudo-terminal may be constructed from a loopback driver, a *tty* module and a control module (Figure 8).

Since Streams are dynamically constructed it is possible to switch between user and kernel versions of a module without rebooting, although existing connections are presumably lost. A virtual terminal protocol is a likely candidate for migrating in this fashion. Figure 9 shows a remote login server configuration with a user mode virtual terminal protocol implementation. (A pseudo-terminal connects the virtual terminal module and the user's process.) In Figure 10 it has migrated into the kernel. Figure 11 shows the client end in user mode and Figure 12 shows its migration.

Taking the kernel's viewpoint we may call the above *immigration*. On the other hand, consider *emigration*. When a low level protocol implementation has been well tested and tuned it may become a candidate for moving onto a dedicated processor. There are various reasons for doing this; it may improve performance, reduce the load on the main CPU(s) or it may be easier to package a product in this manner. (But none should be taken for granted.) Of course STREAMS says nothing about the structure of the hardware involved in a Stream. There is nothing hard and fast about where we "draw the bottom line" in configurations such as that of Figure 2. For example the functions of the *tty* module could emigrate to firmware on the serial interface board. It may be possible to ease this by using mechanisms similar to those of STREAMS in the firmware when it is first designed. A hardware driver or expatriated module on the outboard processor could be given the impression of a Stream connecting it to the lowest level module in the kernel, and *vice versa*. This would be assisted by designing the hardware/kernel interface so that it appears transparent to the modules above and below. Of course these principles would also simplify the *repatriation* of modules if required for further enhancement or debugging. Figure 13 shows a variation of Figure 7 with several low level modules emigrated onto their associated hardware.

If the hardware/kernel interface supports many Streams crossing it simultaneously, an upper multiplexor could emigrate. This would be the case with an outboard TCP implementation. There may be examples of usefully expatriated lower multiplexors. Unless all the network interfaces are accessible by one outboard processor, IP does not look like one of them!

Ideally the location of a module within the kernel or an outboard processor should be transparent to the configuration of Streams. That is, a user process should be able to push and pop modules and link up multiplexor drivers without regard to their implementation. We believe that this is possible above multiplexors but that transparent dynamic linking below multiplexors may require changes to STREAMS.

Given such a mechanism for implementing and configuring the lower modules of a Stream on an outboard processor, we raise the possibility of expatriating middle level modules as well. That is, a module is to be implemented outside the kernel on an outboard processor but is to appear on a Stream between modules that are both implemented inside the kernel. Clearly the transparency of the hardware/kernel interface from the point of view of Streams is crucial to such a scheme. A module doing data encryption is a likely candidate for such treatment as these functions are heavy consumers of both processing and programming resources. Figure 14 shows an encryption/decryption module added to the Virtual Terminal Server of Figure 10.

Of course outboard processor implementations of network protocols are common, including TCP (with IP limited to one network interface). Our interest here is with structuring kernel software and network interface firmware so that migration between them is easy and the architecture of the whole system is preserved.

6. Alternative Systems

The Eighth Edition of the UNIX system (ie. the Bell Laboratories research version) implements Ritchie's streams with some interesting extensions[11] and a Network File System.[12] It is also understood to have synchronous I/O multiplexing support.

The Berkeley Software Distribution (release 4.1 and later) has extensive support for kernel implementations of network protocols, defines a service model for the user/kernel interface, has synchronous I/O multipexing support and has "asynchronous" I/O support using signals.[13]

7. Summary

System V.3 STREAMS offers a basis for development of new computer networking products for System V and for the rationalisation of old ones. The development of terminal oriented services would be assisted if terminal drivers were consistently written or rewritten using STREAMS.

Standard service interface definitions allow some applications to be written in a protocol independent fashion and allow mixing of kernel protocol modules. However these possibilities may not be achieved if such definitions are not developed and published in a timely manner.

The way in that STREAMS passes information in messages rather than by procedure calling lends itself to protocol migration especially between kernel and hardware. Such migration may not be not limited to the ends of a STREAM.

References

1. K. Thompson, ''UNIX Implementation,'' in *UNIX Programmer's Manual, Seventh Edition*, vol. 2, Bell Laboratories, October 1978.
2. D. M. Ritchie, ''A Stream Input-Output System,'' *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1897-1910, October 1984.
3. David J. Olander et al, ''A Framework for Networking in System V,'' *USENIX Conference Proceedings*, Atlanta, Georgia, June 1986.
4. Defense Communications Agency, *DDN Protocol Handbook*, December 1985.
5. Robert M. Metcalfe and David R. Boggs, ''Ethernet: Distributed Packet Switching for Local Computer Networks,'' *Communications of the ACM*, vol. 19, no. 7, July 1976.
6. *DOD Standard Internet Protocol*, Internet Working Group, IEN 128, January 1980.
7. ISO IS 8072, *Information Processing Systems – Open Systems Interconnection – Transport Service Definition*, 1986.
8. D. A. Nowitz and M. E. Lesk, ''A Dial-Up Network of UNIX Systems,'' in *UNIX Programmer's Manual, Seventh Edition*, vol. 2, Bell Laboratories, October 1978.
9. Andrew P. Rifkin et al, ''RFS Architectural Overview,'' *USENIX Conference Proceedings*, Atlanta, Georgia, June 1986.
10. Russel Sandberg, ''Design and Implementation of the Sun Network Filesystem,'' *Usenix Conference Proceedings*, p. 119, Portland, Oregon, Summer 1985.
11. D. L. Presotto and D. M. Ritchie, ''Interprocess Communication in the Eighth Edition Unix System,'' *Usenix Conference Proceedings*, pp. 309-316, Portland, Oregon, Summer 1985.
12. P. J. Weinberger, ''The Version 8 Network File System,'' *Usenix Summer Conference Proceedings*, Salt Lake City, Utah, June 1984.
13. W. Joy et al, ''Berkeley Software Architecture Manual, 4.3BSD Edition,'' in *UNIX Programmer's Supplementary Documents*, vol. 1 (PS1:6), Computer Systems Research Group, University of California, Berkeley, May 1986.

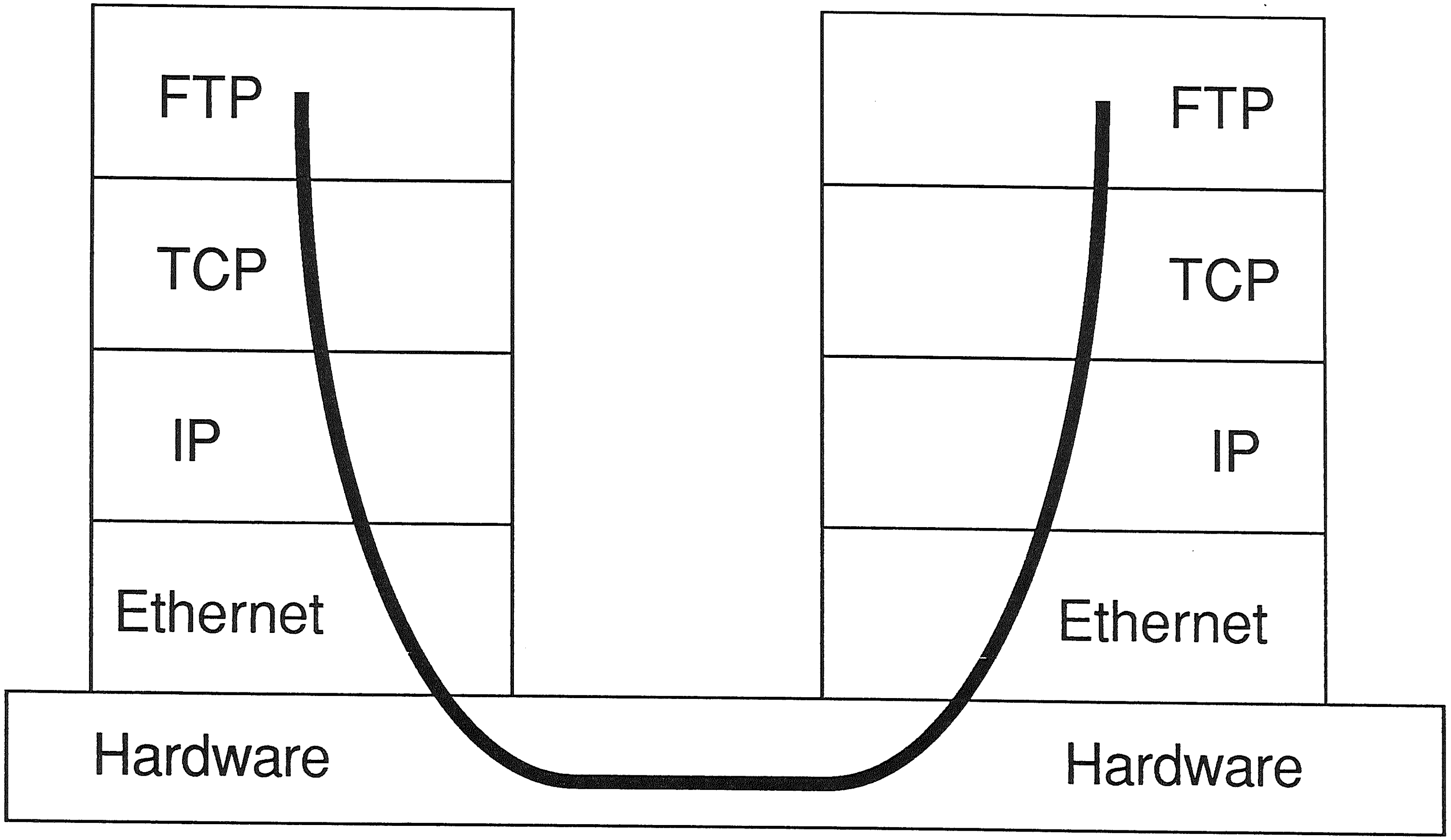


Figure 1

Data Flow through Protocol Stacks

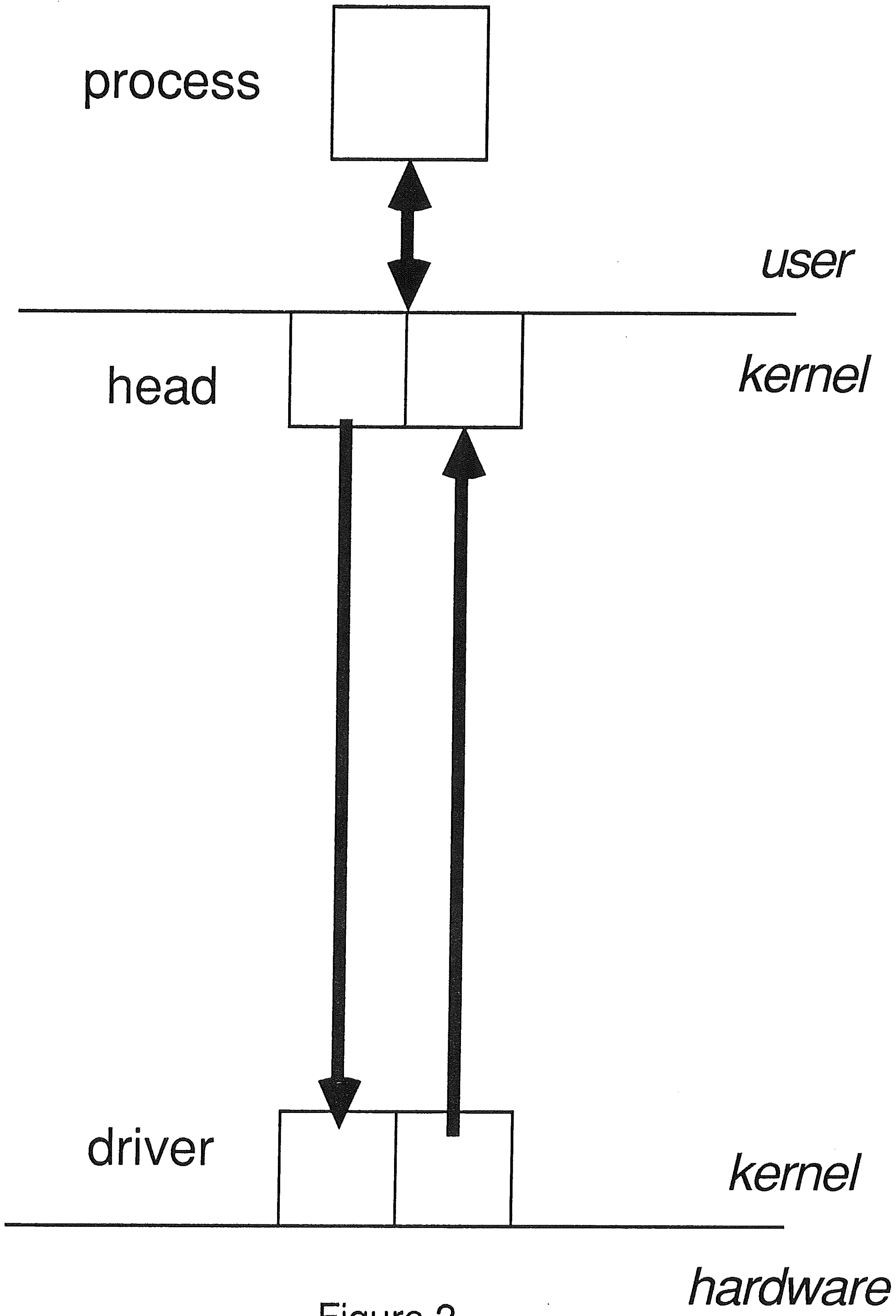


Figure 2

A Simple Stream

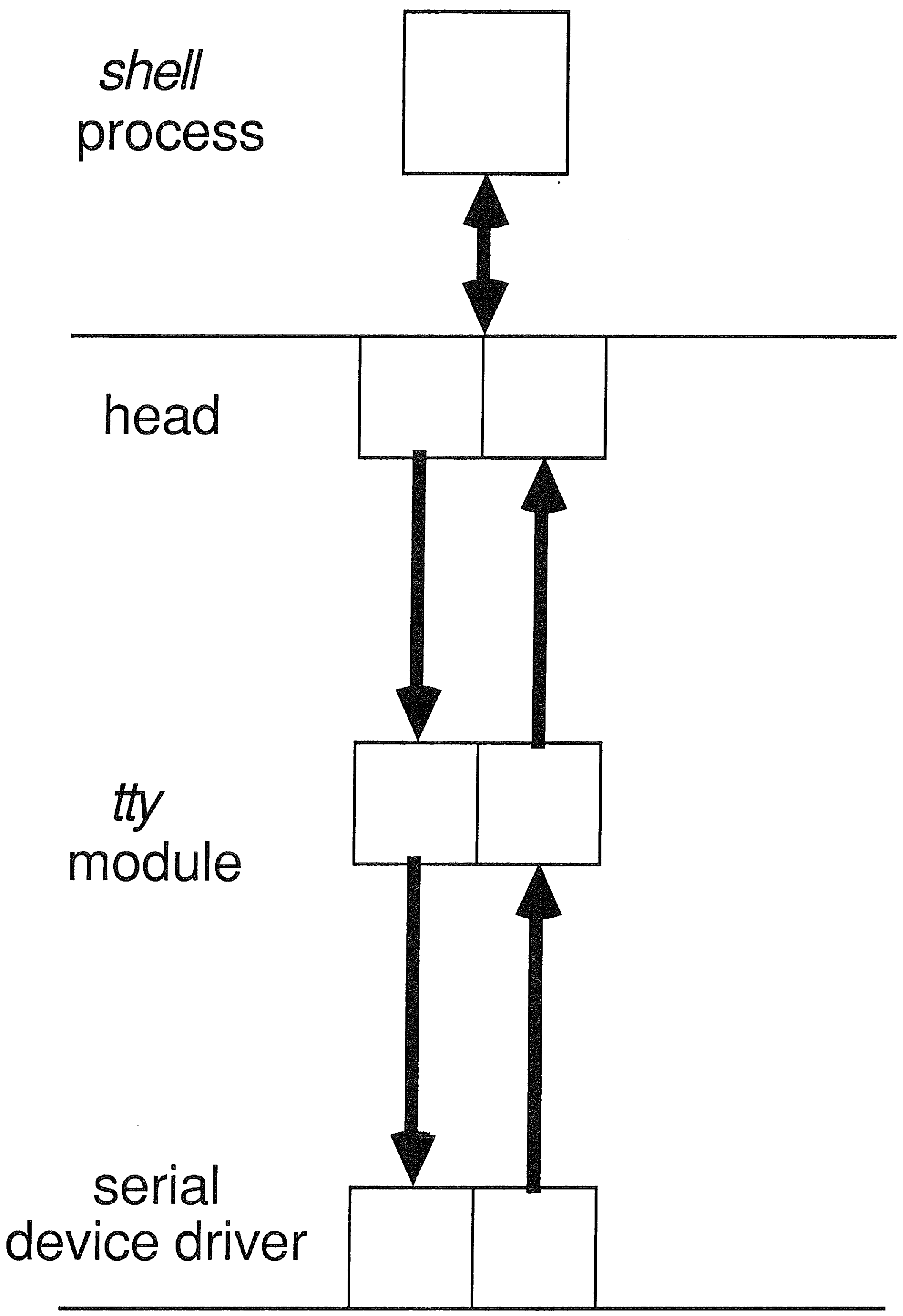


Figure 3

Pushing a Module

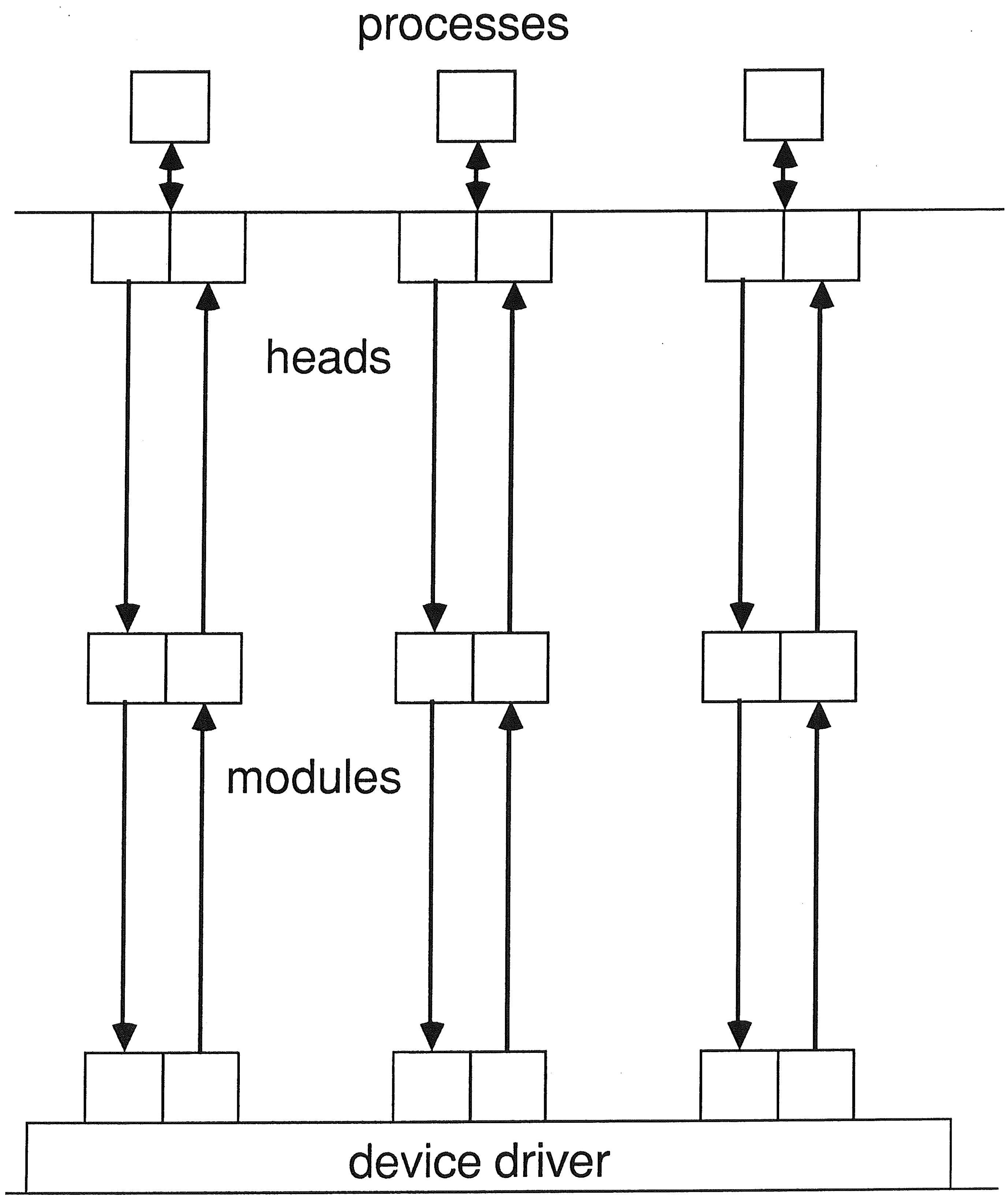


Figure 4

A Multiplexing Driver

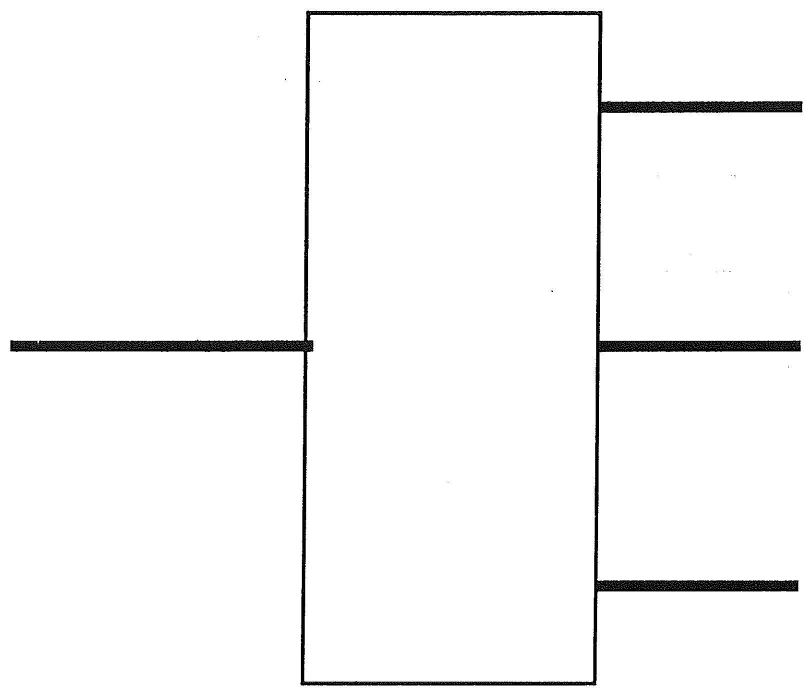


Figure 5
Upper Multiplexing

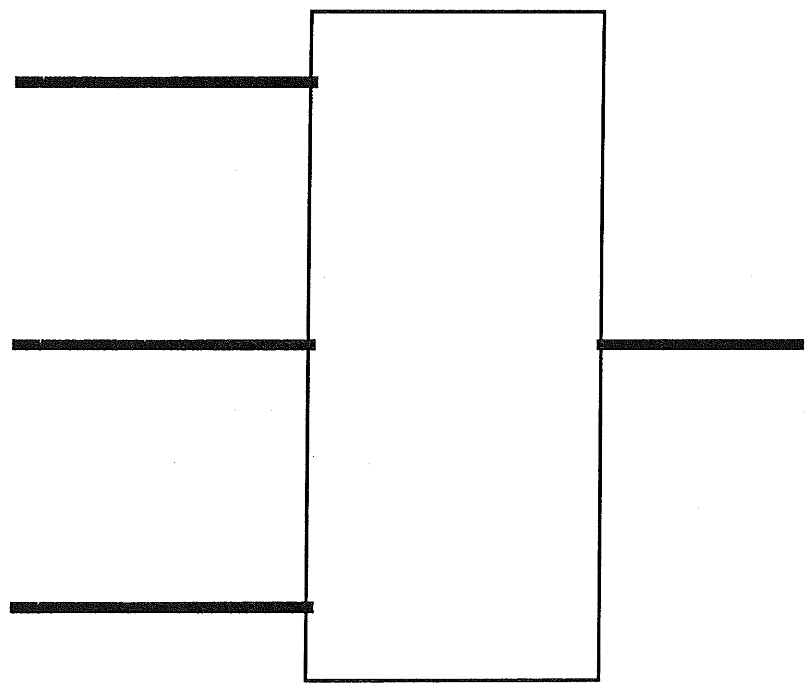


Figure 6
Lower Multiplexing

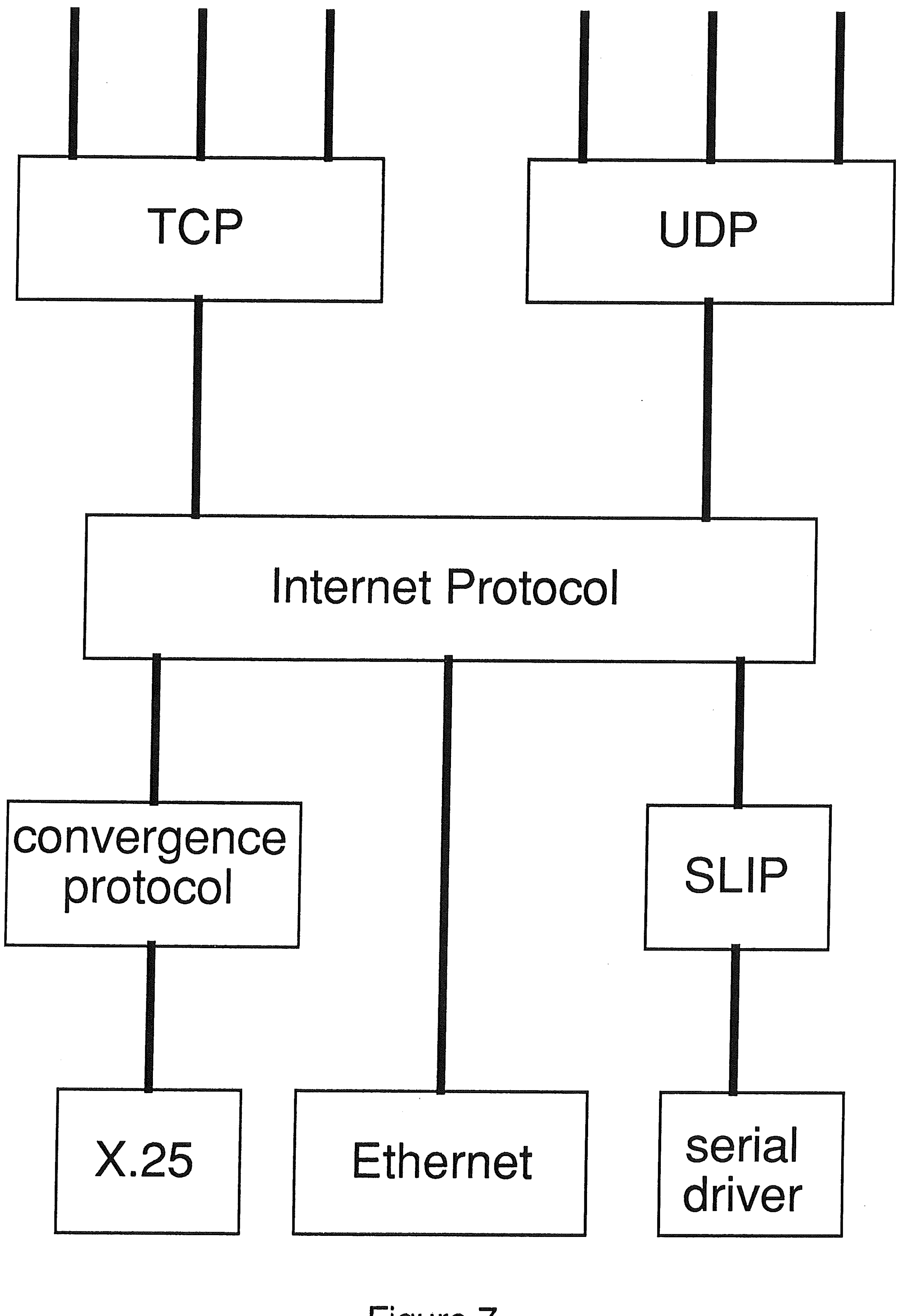


Figure 7

An Upper and Lower Multiplexor

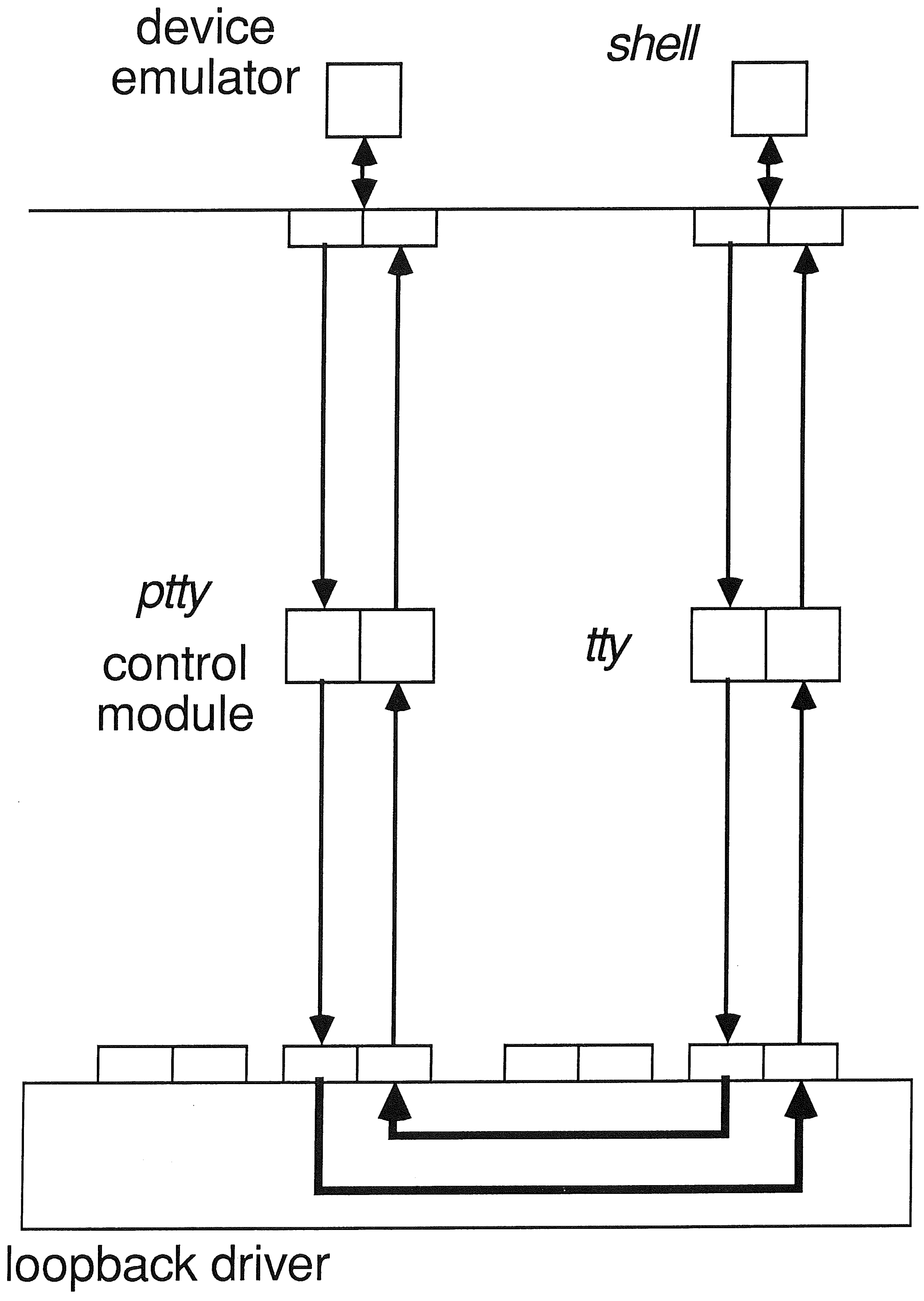


Figure 8

Construction of Pseudo-Terminals

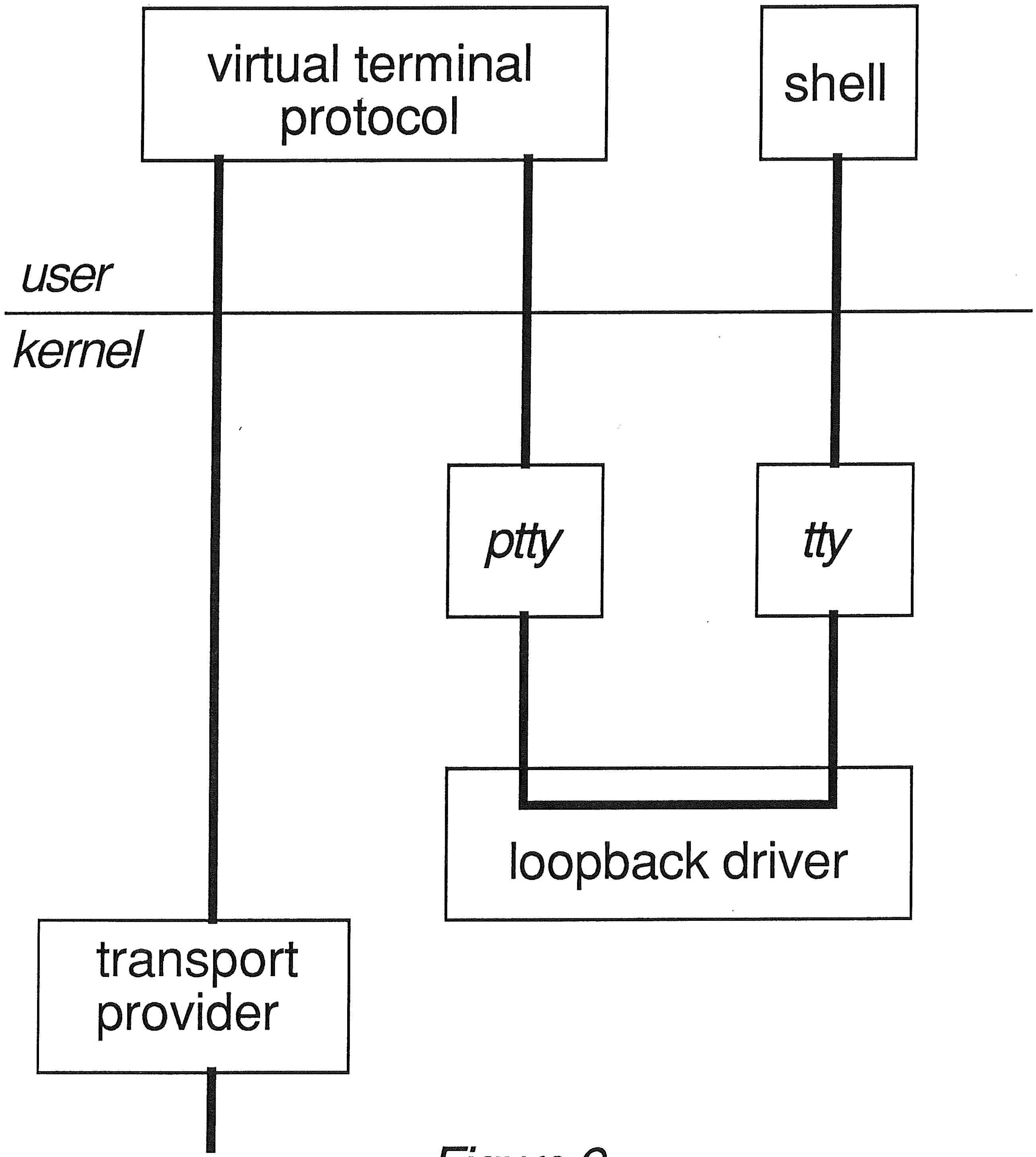


Figure 9

Virtual Terminal Server

User Mode Implementation

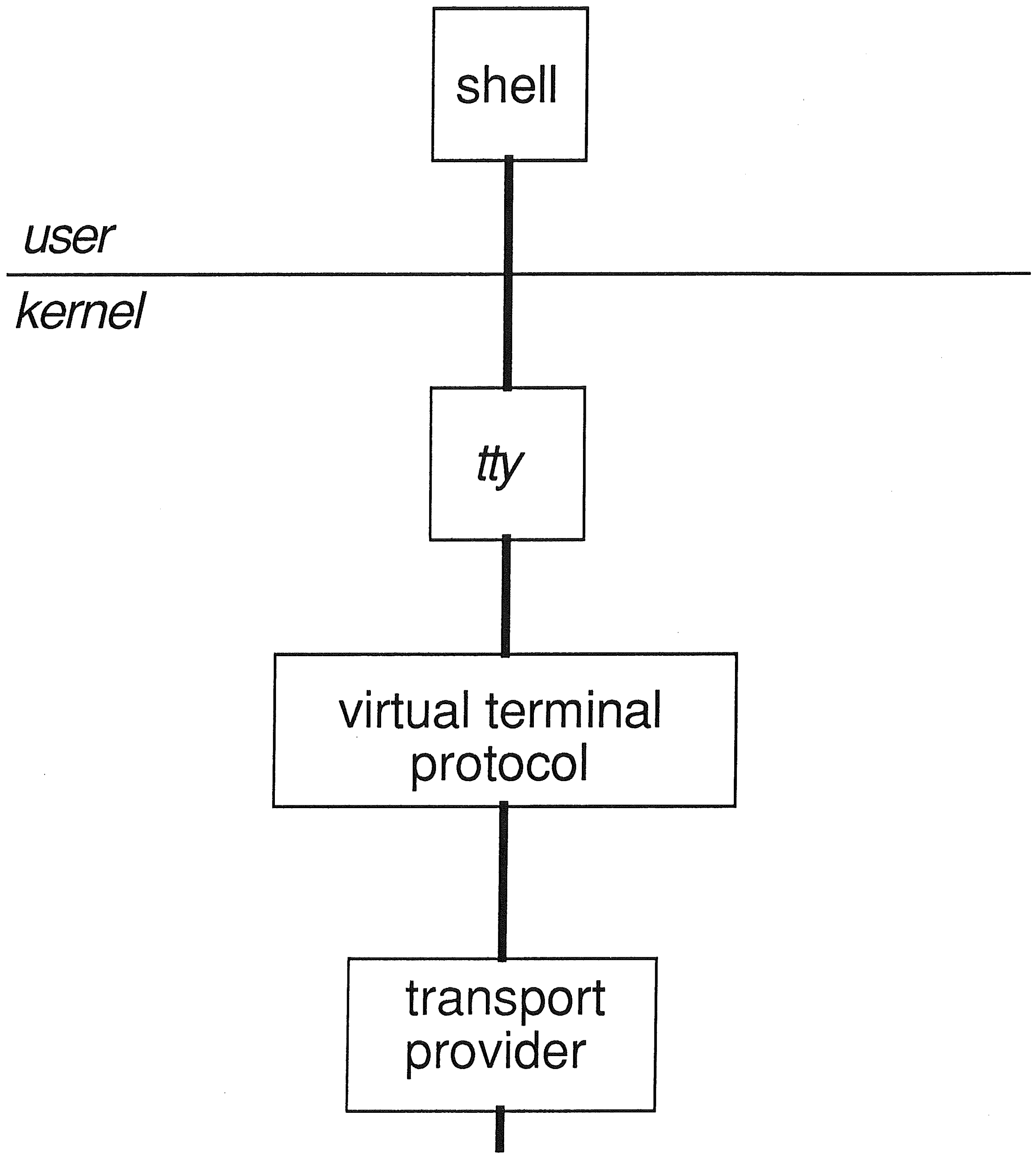


Figure 10

Virtual Terminal Server

Kernel Mode Implementation

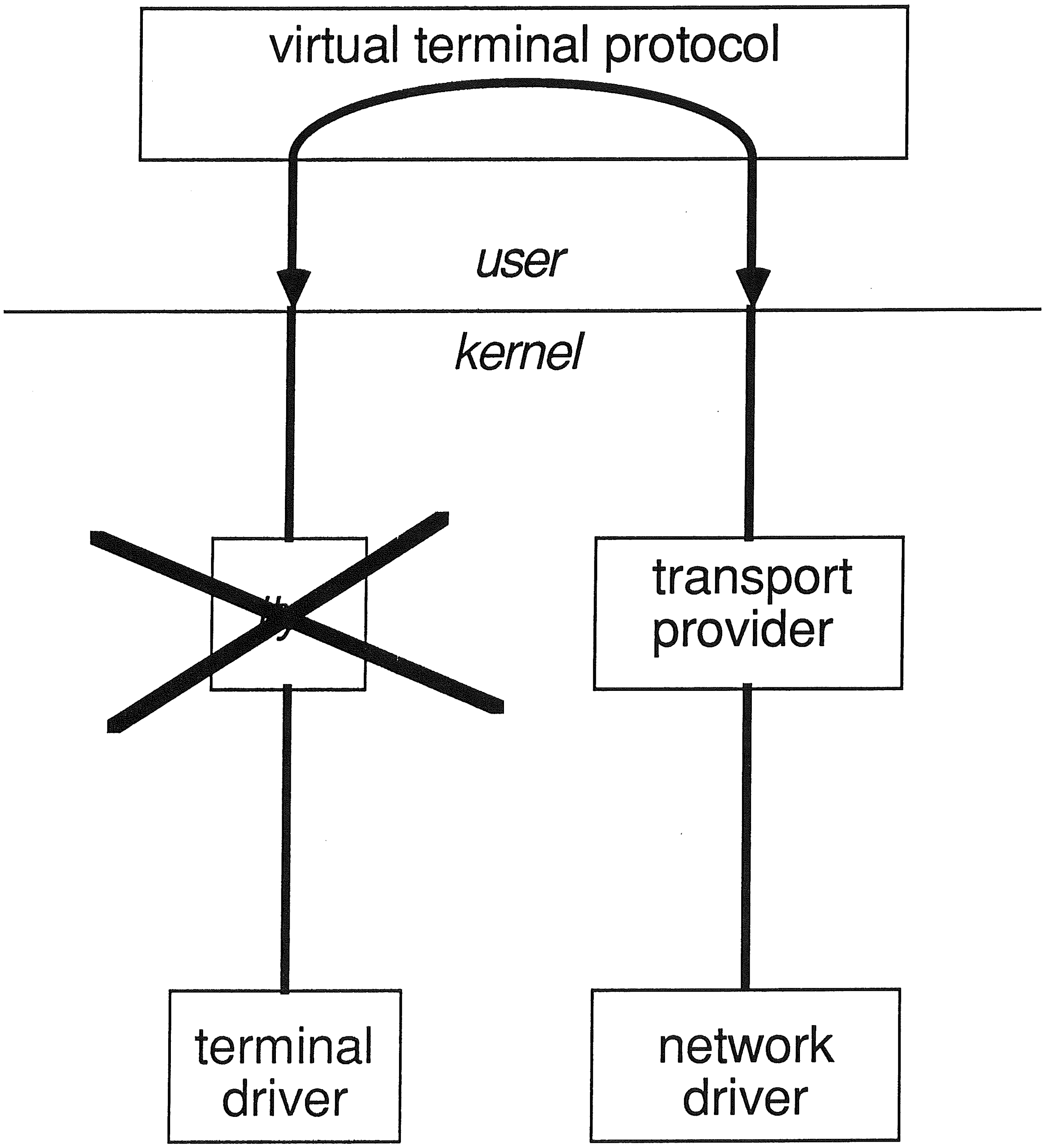


Figure 11

Virtual Terminal Client
User Mode Implementation

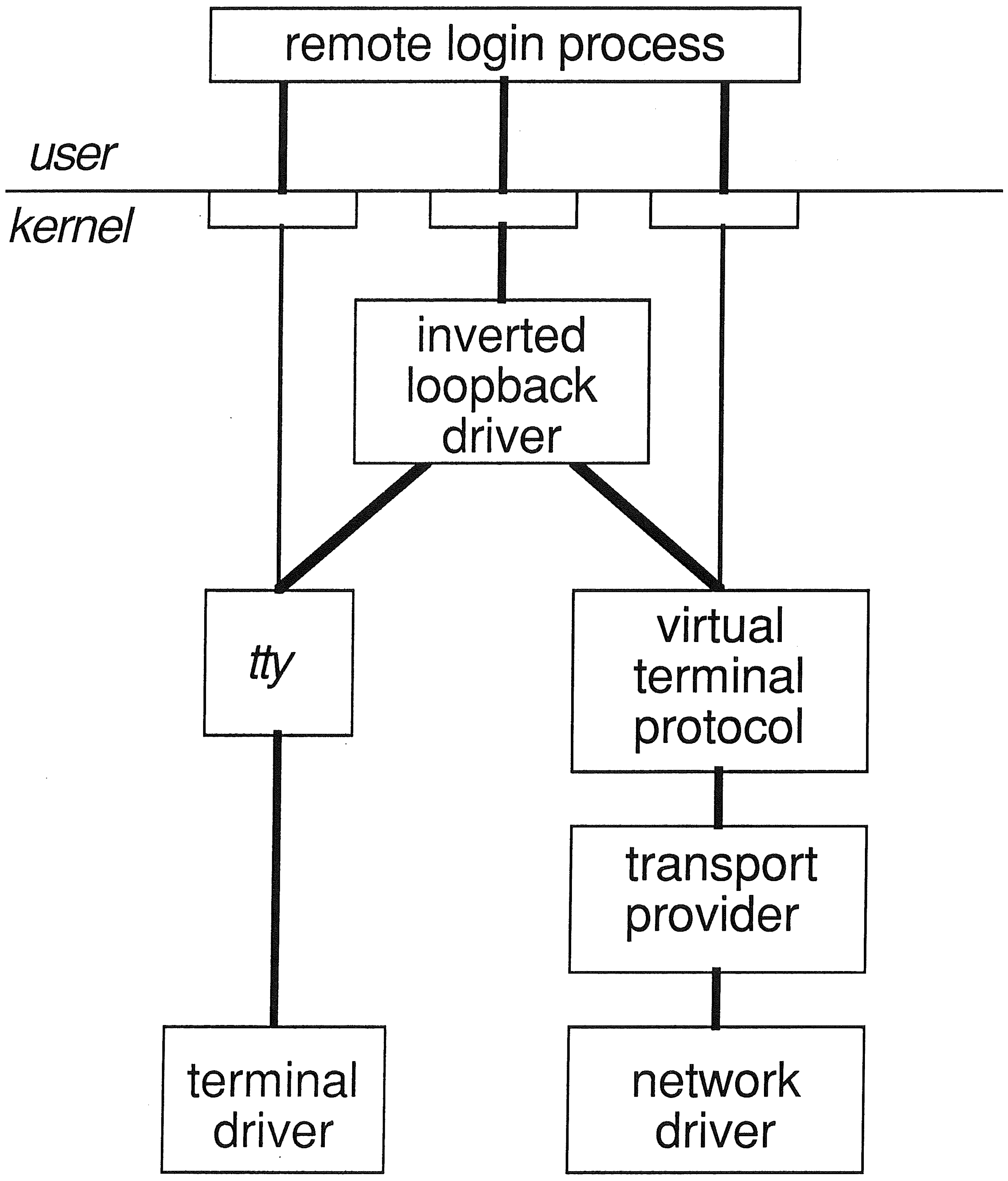


Figure 12

Virtual Terminal Client

Kernel Mode Implementation

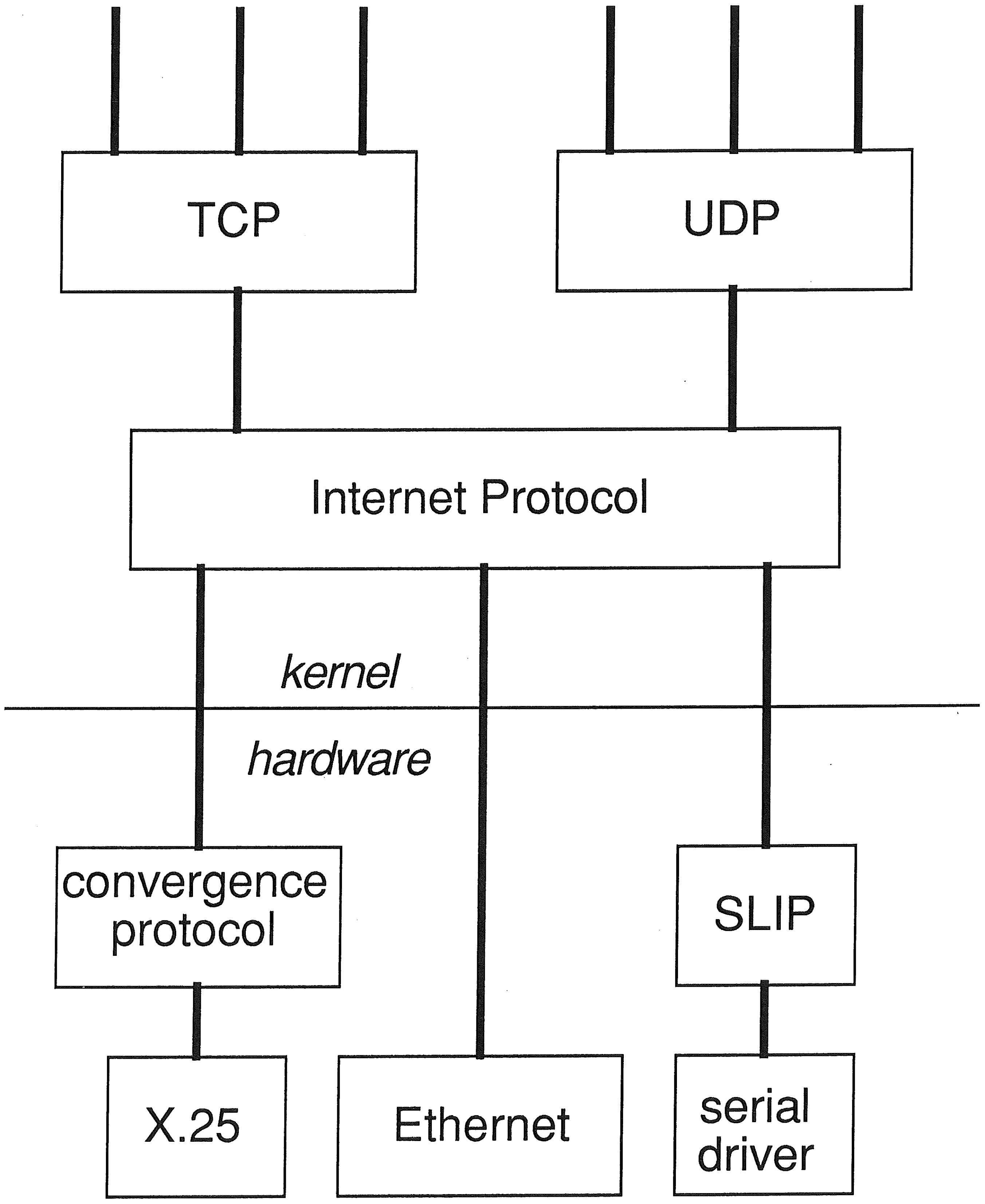


Figure 13

Low Level Protocol Emigration

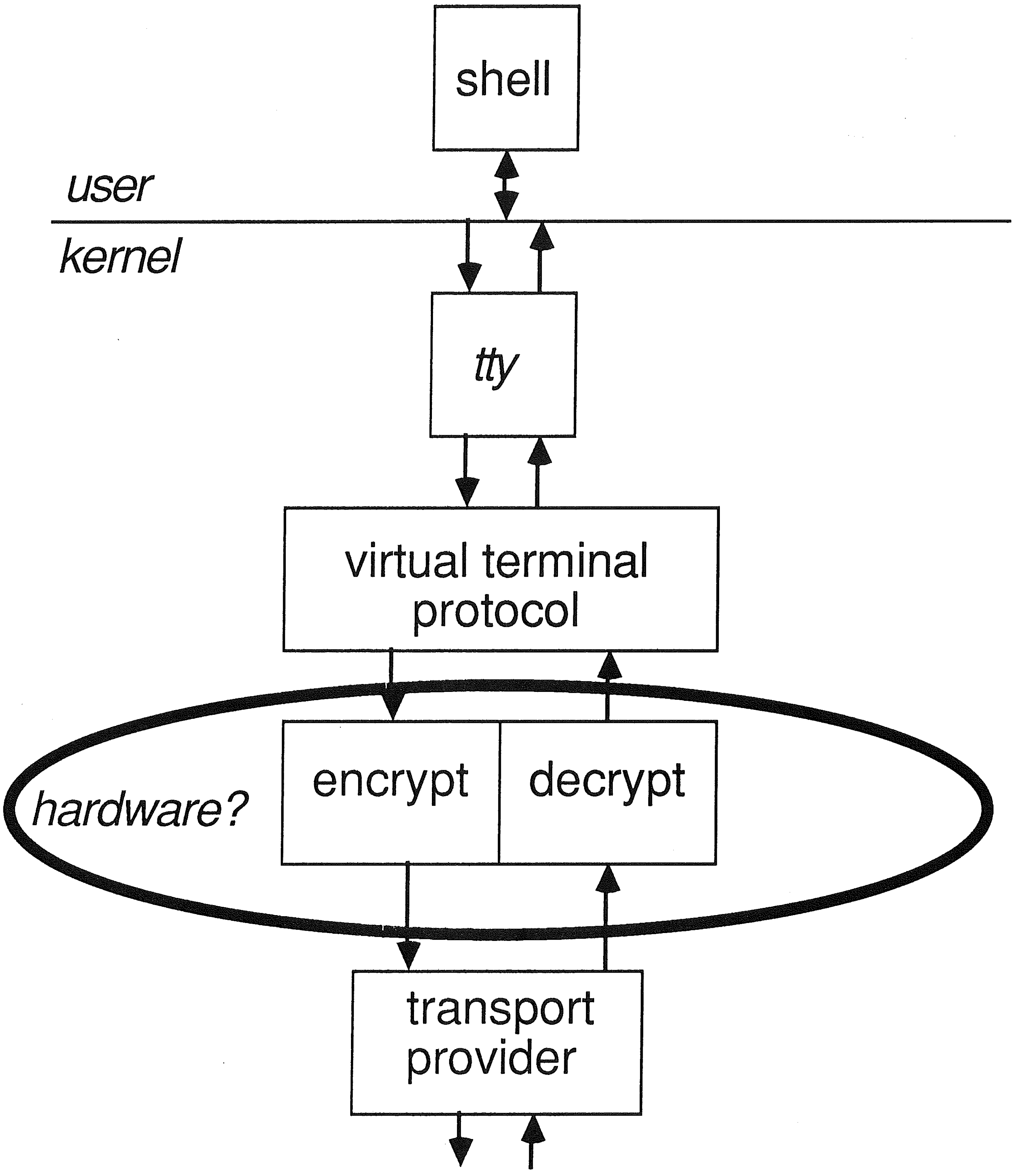


Figure 14
Virtual Terminal Server
with Data Encryption

Optimizing C: Benchmarks and Real Work

Michael Tilson

Many UNIX systems are now supplied with "globally optimizing" C compilers. These compilers perform a wide range of program transformations, such as common sub-expression elimination or loop invariant removal. These transformations can considerably improve the speed of benchmark programs (such as the ever popular "Sieve"), and they improve the speed of certain classes of applications programs.

However, users have been disappointed to discover that optimizing C compilers make little or no improvement to heavily used UNIX utilities or to the UNIX kernel itself.[1] Many parts of the UNIX system have been hand-tuned over the years; the opportunities for traditional mechanical optimization are limited since the C language allows the programmer to perform, at source level, optimizations such as register allocation and indexing strength reduction via pointer variables.

This talk will discuss current work at HCR aimed at using an optimizing C compiler to improve the performance of real UNIX utilities. The talk will review the state of optimizing C compilers, performance that can be expected on various classes of program, variations with computer architecture, and the issues involved when attempting to boost the performance of already well-tuned utilities.

Optimization will be shown as a "programmer productivity" issue. In C at least, almost any degree of optimization can be achieved by hand. Developers of optimizing compilers should look for high-payoff transformations that would be difficult to perform by hand, or that would destroy the readability and structure of the program if performed at source level.

[1]With some compilers, you are lucky if the program continues to work after "optimization", especially in the case of the UNIX kernel. The C language presents optimization difficulties, especially with global variables, pointers, and asynchronous events, and some suppliers are not sufficiently careful. The UNIX "signal" mechanism ensures that even user-level programs can and do process interrupts, so an optimizer that can't compile the kernel also shouldn't be trusted with user-level code. But if a compiler is careful to preserve the proper semantics, then the job of improving performance becomes even more difficult.

MEASURING DATABASE PERFORMANCE USING THE TP1 BENCHMARK

Ken J. McDonell

Department of Computer Science
Monash University
Clayton, Victoria 3168, AUSTRALIA

Acsnet: kenj@moncsbruce.oz

ABSTRACT

This note reports on some performance experiments conducted with a commercially available relational database management system (let's call it DBMS-R) in conjunction with the TP1 benchmark1. These tests are of particular interest given the popularity of TP1 as a *de facto* standard for measuring on-line transaction processing throughput. This paper assumes the reader is familiar with the TP1 benchmark; full details may be found in1.

In all cases, the tests were run on an unloaded Unix[1] machine in multi-user mode (with the usual assortment of daemons, especially the Ethernet ones). Several Unix machines were used, all from the one vendor's model range; they shall be referred to as Model-1, Model-2 and Model-4 (model numbers crudely approximate to relative raw performance).

Except where stated to the contrary (for some Model-4 tests), the filesystems all had a default configuration with a block size of 16K bytes.

The same brand of disk drives was used in all tests.

This paper appeared in AUUGN Volume 8 Number 3-4.

AUUGN Editor

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

Database Management Systems: Efficient Implementation for UNIX systems

Angela Heal BSc
Queensland Department of Primary Industry

The recent past has seen the rise in importance of Database Management systems as UNIX utilities. In this paper we examine some techniques which may be used to implement an efficient and reliable UNIX DBMS. The centralised and noncentralised approaches and the system resources that they require in the area of concurrency control are discussed. Several commercially available DBMS's are explored and the techniques used in them discussed. An analysis is made of the case where concurrency control is implemented by using an undo log and record locking. Here, providing a minimal level of interprocess communication (the fifo or named pipe) is available, the centralised approach offers many advantages over the non-centralised. This approach makes it possible to create a fast and reliable DBMS on a UNIX system and still retain a high level of portability. Decreasing the portability of the communications module makes it possible to substantially increase the throughput of such a system without significant rises in system complexity by exploiting version specific features such as shared memory.

What's in a name? (or coping with lots of small files)

J. Lions

University of New South Wales, Kensington, NSW 2033.

ABSTRACT

The UNIX System provides both tools and guidance for carrying out particular tasks. It encourages users to organise their data as lots of small files, and, for program developers, it provides strong guidance for ways to organise these files. For other tasks that can also be effectively organised using many small files, the tools are provided, but the guidance is not nearly so clear or effective.

This paper describes one attempt to manage several hundred small files for an 'office' application where the main complicating factor has been the sheer number of files. These have been arranged in a directory hierarchy up to four levels deep and containing approximately more than eighty subdirectories. Certain major subdirectories have been designated as 'areas', and some simple, effective tools have been developed to allow convenient 'navigation' between areas. Further developments have been made to the original concept, e.g. to migrate many area-specific procedures from the usual bin directory to the various area directories.

Keywords

data files and directories, UNIX system commands, office automation

CR Categories

E.5, H.3.2, H.4.1, I.7

Note

This paper is to be presented at the Australian UNIX systems User Group meeting in Sydney on August 28, 1987.

1. Introduction

The UNIX* system has many facets. Most obviously it provides a kernel program plus many useful user level programs (software tools) including filters and command interpreters that can be assembled into interesting and versatile combinations. More subtly, UNIX provides some strong models for organising programs and data in particular ways, and gives strong guidelines for carrying out many computational tasks. In so doing the UNIX System expresses a particular 'bottom-up' philosophy for organising and arranging computations (a philosophy that has won many disciples and converts).

The UNIX System appeals particularly to program developers. (This is hardly surprising in view of its origins.) By providing strong conventions for arranging files into directories, for naming files and directories, and through such commands as make, ar, etc., it provides program developers with a clear model as to how they should use the system (e.g. choose file names, arrange directories, use PATH to control the selection of executable files, etc.). However the system does not provide such a clear model for some other types of application, especially where files proliferate and do not seem to fall into neat categories. At this point many people rush out and buy a conventional DBMS package, but that isn't always the real solution. The unadorned UNIX system with its support for handling many small files economically does provide a suitable base system for many data applications. What the standard system does not do, in my experience, is give strong hints how this should be done.

This paper describes some of the conventions that I have established for my own use and the software tools that have resulted. They are not particularly spectacular (it is possible that a wholesale conversion to emacs might be an even better solution, but it is not clear that it would — especially on heavily loaded machines such as our VAX system — and of course emacs is not universally available). Although the basic approach and tools are now stable, I haven't yet run out of ideas for extending and refining them. The approach may be summarised as follows:

1. Devise a suitable hierarchy and associated directory tree for the storage of data files. This is not necessarily easy but the aim will be to make an obvious place for each possible file, and to cluster related files close together in the directory tree;
2. Designate major subtrees as *areas*. Give each of these a distinctive name and associated code-letter. The areas will generally, but need not necessarily, be distinct and non-overlapping. Moreover, some subtrees may not belong to any area (but experience suggests that these should be as few as possible!);
3. If the number of files in an area is large, the area directory may contain subdirectories which contain the data files. The location of a file in a particular subtree can then be used to convey status information;
4. Give files short, convenient names (e.g. generated mechanically) rather than particularly meaningful names which may be difficult to devise. For each area, devise an *index file* which, for each file in the area, contains a record relating the file name (relative to the area directory) with a string of meaningful terms;
5. Provide convenient commands to skip from one area to another. Set up procedures (using grep) to search index files for particular terms or strings, and hence to select files for editing;
6. Procedures will be needed to carry out functions for each area (e.g. to create the index or a new data file). Some of these will be general, but others will need to be tailored for individual areas, or will only apply to particular areas. Locate such procedures within the area's base directory (i.e. not within a more normal bin directory), but use them indirectly, i.e. via other procedures that are located via the normal bin directory or equivalent.

---

* UNIX is a registered trademark of A.T. & T. Bell Laboratories.

2. *A Solution to a Problem*

This paper describes a solution to a real problem. As a journal editor for some years, I had to correspond with numerous people about articles offered for publication, books offered for review, etc. The average 'life-time' for correspondence on a particular item is over a year, and more than one thousand items of outgoing correspondence are generated each year. After experimenting with a number of different arrangements (see Lions, 1985), a firm decision was made to establish a separate data file (*item activity file*) for each item (article or book). When correspondence relating to an item is needed, the corresponding item activity file is edited to add letter-generating commands. These commands lie dormant until despatch time draws near. Then a special command (take) is executed to find all the recently modified item activity files, and to generate the actual letters in a batch. (Take is a shell script that uses find, nroff with special macro files, a special filter program, etc.)

The names for item activity files consist simply of an alphabetic character followed by a serial number, e.g. a347, b545, with serial numbers simply being assigned sequentially in chronological order. (This highly unoriginal practice was established long before the computer system.) The alphabetic character a is used for all articles, and b, for all books. When later the need to establish individual files for people also became apparent, by some arcane reasoning, the letter c (for *colleagues*) was chosen to prefix the serial number for such files.

Files relating to items of the same type originally appeared in the same same directory, i.e. level one subdirectories (of the editor's directory tree) called *articles*, *books* and *colleagues* have existed 'always'. As the number of files grew, these directories soon became uncomfortably large. Level two directories were soon established, and the item activity files were demoted down a level. Assignments to the level two subdirectories can be made in various ways, but the most useful way is to cluster items so that those at the same stage of processing appear together, so that an item's presence in a particular directory reflects its current state. The obverse side to this is that items migrate between level two directories during their life-times, and hence they do not retain a single path-name forever.

Imagine that two letters have just been received and require acknowledgement: a review for a book called 'Denotational Compilers' by Simonsen (or should it be 'Simmonson'), and a report from a referee on article on 'Information Hiding'. Clearly, if the serial numbers for the two items are already known, the associated item activity files can be found as soon as their status (and hence level two subdirectory) is determined. However, if the serial number is not known, then the available information needs to be translated into the file name, with any ambiguities resolved along the way. Clearly this is a task well-suited to a computer, and can be achieved very effectively using grep applied to purpose-built *index files* that relate file names to a string derived from the item title and author name(s) for individual areas. Sample index records might be:

```
current/a347:      information hiding      mccarthy
ready/b545: denotational compilers  simonson
```

(File names are relative to the area directory and terminated by a colon. For simplicity, all letters are in lower-case.) All the details of searching, etc. have now been embedded in a shell script called edit (that eventually invokes vi), so that to start an editor for the book's activity file, any one of the following commands with suffice:

```
edit -b 545
edit -b mons
edit -b compile
```

The first argument, -b, specifies the area, and the second, a search string for the index. The first argument may be omitted if the *current area* is already books. Which one of the three alternatives is selected at a particular time will depend on the information readily to hand. If the serial number is known, it is always the best choice for the search string (second argument) because it should be unique. If it is not known, then a string derived from the title or author name(s) can be used, but it may not be unique, and the user may have to supply (interactively) a second search string.

If, while editing the book file to prepare a 'receipt and thank-you' letter for the reviewer (name 'Robinson'), it is apparent that the reviewer's address needs changing, the command:

```
        edit -c binso
```

can be interpolated into the editing session so that the file recording Robinson's vital statistics can be found and modified. If the wording of letter needs refining, the command:

```
        edit -m receipt
```

can be used. What is m? It stands for macros, which was always a level two directory, but only was made an area some time ago once the potential usefulness of the above command became apparent.

3. Areas

An *area* is a collection of files and directory that form a sub-tree of the main directory tree. The distinguishing feature of an area is the existence of a particular type of executable file in (my) bin directory. For example, for the books area, there exists a file bin/b that contains:

```
        #!/bin/sh
        AREA=$HOME/books
        export AREA
        cd $AREA
        PS1='<b> '
```

This procedure is designed to be executed 'in-line' by the Bourne shell. The 'current area' (remembered as the variable AREA) and the current directory are changed to the books area directory (the shell prompt is also adjusted). The procedure may be invoked directly (via a command '. b') to change the current area semi-permanently (e.g. after all the articles have been processed, and it is time to deal with the books). It can also be invoked indirectly, and temporarily, via the command bin/go:

```
        #!/bin/sh
        # change area - exec'd in line
        case $1 in
        -a)    . a ; A="articles" ;;
        -b)    . b ; A="books" ;;
        -c)    . c ; A="colleagues" ;;
        -d)    . d ; A="documents" ;;
        -g)    . g ; A="general" ;;
        -h)    . h ; A="home directory" ;;
        -j)    . j ; A="journal" ;;
        -l)    . l ; A="log" ;;
        -m)    . m ; A="macros" ;;
        -r)    PWD=`pwd` ; export PWD
               . r ; A="reviews" ;;
        -s)    . s ; A="suppliers" ;;
        -x)    . x ; A="bin directory" ;;
        -?)    echo unknown area $1 && exit 3 ;;
        esac
        case $1 in
        -?)    [ -t 0 ] && echo $A ; shift
        esac
```

This procedure, which is called in line by e.g. edit, contains an entry for each of the twelve areas used in the editorial application. With a little ingenuity, it has been possible to find a unique letter to associate with each area. There are a number of special cases that can be mentioned in passing:

1. Book suppliers have both the general property of receiving correspondence and the special property of receiving certain types of correspondence. As a result, it has been convenient to have the suppliers area as a sub-area of the colleagues area.

2. The home area corresponds to the home (i.e. level one) directory. It is useful occasionally to have this.

3. The directory for the area designated by x is simply the bin directory, which contains executable files. Most of these are shell scripts, which need to be adjusted (edited) occasionally. Where the executable file is actually a compiled *C* program the index entry points to the corresponding entry in the src directory. Thus the bin directory also has an index file that contains entries such as:

   ```
   take: generate batch of letters
   mk:   miscellaneous things to do in area directory
   ../src/enter.c:   update fields in pattern file
   ```

   Take has already been mentioned; mk is described below; and enter happens to be a simple data-entry program that is used for creating new data files.

4. Area-Related Procedures

Possibly the most successful outcome of the formalisation of areas has been finding suitable homes for a variety of special purpose shell scripts. Activities such as creating new data files, or recreating the index files, in different areas have many things in common, but also have individual differences. To create a new data file in the current area, there is a procedure new in the bin directory that invokes a procedure called newpart that resides in the area's directory. This in turn may call enter (that lives in bin) and uses a data file called pattern (that lives in the area's directory). Thus there is a uniform way of calling one of a set of area dependent procedures.

Special purpose shell scripts that apply to only one, or a few, areas used to clutter up the bin directory and naming them appropriately was a problem. (Another problem was to find them again if they were used only infrequently!) To make a long story short, a command mk has solved this problem quite successfully. It takes an optional first argument to select the area of interest, and another argument to select a procedure from a 'super shell-script'. The latter occurs in a file, mk.file, located in the area directory. The code for mk is quite simple:

```
#!/bin/sh
# mk - operates on a file called mk.file
# located in the current directory or above.
. go  # change areas if -a argument
for j in 1 2 3 ; do
      [ -s mk.file ] && break
      cd .. ; pwd # climb tree ...
done
if [ $1 ] ; then option=$1 ; else
      grep '##' mk.file  # show choices
      echo 'enter option (or q): \c'
      read option
fi
case $option in
q)    exit ;;           # quit
*)    sh mk.file $option $2* ;;
esac
```

After changing areas, a mk.file is sought in the current directory or above. If no option is specified on the command line, all lines in the mk.file with the pattern '##' are displayed, and the user is asked to make a choice. A sample mk.file is as follows:

```
# @#$:      miscellaneous things to do with macro files
. m
case $1 in
gpsmacs)      ## combine & strip gmacs, pmacs & smacs
      wc gmacs pmacs smacs &
      sed ' /^\.\\"/d
            s/\\".*//
            s/[   ][    ]*$//
            s/^\([\.'"'"']..\) /\1/
            s/ \.ds \(..\)/ .ds\1/
      ' gmacs pmacs smacs > gpsmacs
      wc gpsmacs
      ;;
symbols)      ## list all nroff symbols used
      uncompress mksymbols.Z
      mksymbols
      ;;
index)              ##
      rm -s temp*  */temp*
      echo 'index:' > index
      grep '@#\$:' * */* \
      |  sed 's/:.*@#\$:./:   /' \
      |  sort  >> index
      ;;
*)    echo $0: Option '[' $1 ']' not recognised. ;;
esac
```

It will be seen that the file consists almost entirely of a shell case statement, with four entries. The first three, which represent valid options, have '##' lines. (The first option, for anyone who is interested, shows a script for stripping redundant blanks and comments from nroff/troff files.)

The third option is to recreate the index file. For the original areas, which contain item activity files that conform to a prescribed pattern, the index is generated according to fixed rules. However for other, less disciplined areas, an alternative method is needed. My solution is, for each file that is to be entered in the index, to embed an identifying comment line containing the pattern '@#$:' before the meaningful part of the comment. (See line 1 of the above mk.file.) Such lines can then be sought using grep and edited to create the new index, as shown.

5. *Conclusion*

The idea of formally designating subdirectories as 'areas' and providing suitable support mechanisms has been canvassed. These mechanisms have greatly improved the usability of the UNIX system for at least one office application. It has been found that the area concept can be stretched well beyond the original idea.

6. *Reference*

Lions, J. (1985): The Development of a Correspondence Package, *Australian Computer Journal, 17,* pp. 131-135.

An Image of The Future

Julian Day
Microprocessor Applications

ABSTRACT

In the United States much research has been carried out to solve the complex solution needed to integrate images, text, alphanumeric data and other forms of unstructed data into a database management system using a diverse collection of specialized hardware and software components.

This paper discusses the emergence of an Extended Data Processing (XDP) environment which has been jointly developed by Plexus Computers and Informix Software as a database to store, retrieve and manipulate very large datafiles, including scanned images of very large documents.

Such an environment has extensive applications throughout industry and government, allowing all types of data, image and text to be stored on high-density optical disk.

The XDP system merges a departmental computer and relational database management system with personal Workstations, document scanners, omnifont optical character readers, facsimile machines, laser printers, optical disks, an optical "Jukebox", microfilm and microfiche scanners, plus support for industry-standard network communications protocols.

Already, two large contracts have been taken in the US for such systems. One of these, US WEST DIRECT, is managing a Yellow Pages publishing operation with the system combining images of advertisements along with numeric and text copy into a mixed-mode database.

Plexus is represented in Australia by Microprocessor Applications who are currently identifying potential target markets in industry and government.

This abstract and following information copied from a poor reproduction caused by facsilime transmission. Apologies for any typographical errors.

AUUGN Editor

PLEXUS XDP SYSTEM PRODUCT BACKGROUND

Plexus Computers, Inc
3833 North First St.
San Jose, CA 95134
(4080 943-9433

Microprocessor Applications Pty Ltd.,
Suite 2, 156 Military Road,
Neutral Bay N.S.W. 2089
(02) 908 3666

The Plexus XDP Extended Data Processing System is the first comprehensive commercial computer system optimized for "mixed-mode" data processing. The XDP System integrates database management capabilities with a diverse collection of specialized hardware and software components required to manage various modes of information, including images, text, alphanumeric data and other forms of unstructured data. It solves complex data processing problems with a system that is far more flexible and accessible to end users than previous commercial computer systems.

The XDP System merges a departmental computer and relational database management system with personal computer workstations, document scanners, omnifont optical character readers, facsimile machines, laser printers and optical disks into a comprehensive mixed-mode data management system. It captures, stores, catalogues, manipulates and retrieves many different kinds of information -- from text and alphanumeric data to images and other forms of unstructured data.

Plexus P/95, P/75 and P/55 computers act as data management hubs in XDP Systems, providing powerful but flexible data processing capabilities for a variety of system configurations integrating numerous industry-standard components.

Plexus has signed a $6.6 million research and development contract with the PruTech Research and Development Partnerships to develop and bring to market the next generation of its recently announced XDP System DataServer. The R&D-funded XDP DataServer will be designed as a specialized mixed-mode processor.

One of the company's first customers for the XDP System, US WEST Direct, is managing a Yellow Pages publishing operation with the system. The system combines images of advertisements along with numeric and text copy into a mixed-mode database.

The integration and automation of these various modes of information is expected to pay for itself in its first year of operation and substantially improve customer service and production turnaround time.

Additional target markets for the Plexus XDP System include other telecommunications companies, the military services, federal and state governments, pharmaceutical and health care companies and other organizations requiring solutions to complex data processing problems.

Plexus, with more than 2,200 computer systems installed worldwide, is recognised as an innovator and leader in providing industry-standard computing solutions. With 70 percent of its customers already performing database-intensive applications, the company is well-positioned to extend its product family to provide mixed-mode data integration for commercial systems applications. Additionally, the company's widely imitated multiprocessor architecture for UNIX super-microcomputers, which delivers minicomputer-level performance, is an ideal platform for mixed-mode XDP System applications.

XDP SYSTEM ARCHITECTURE

Plexus provides Extended Data Processing System capabilities in a variety of configurations. The XDP System consists of two major hardware elements managed by Plexus' XDP Operating Environment software.

The hardware components are an XDP DataServer and a network of XDP WorkStations.

Plexus super-microcomputers are ideally suited for the XDP DataServer role, because they employ a powerful multiprocessor architecture. The Plexus P/95, P/75 or P/55 can be used as an XDP DataServer.

The DataServer uses optical and high-speed magnetic disk technology for data storage and management. Overall system storage capacity can be expanded to up to 280 gigabytes with an optional optical disk "jukebox".

The XDP WorkStation is an IBM PC-AT-compatible microcomputer enhanced with an image processing board, interfaces for a variety of peripheral products, and communications software. XDP WorkStations communicate with the XDP DataServer over an Ethernet local area network.

XDP SYSTEM SOFTWARE

The XDP Operating Environment contains XDP DataServer and XDP WorkStation software optimized for mixed-mode data processing. The XDP DataServer software operates under UNIX Sys5, while XDP WorkStation software executes under Microsoft Windows running MS-DOS 3.1. Xdp DataServer and XDP WorkStation software includes standard and extended components designed to allow the development of custom applications for mixed-mode data processing.

XDP DataServer software includes the XDP DataManager and XDP Application Utilities Modules. The XDP DataManager module integrates the Informix-SQL relational database management system with Plexus-supplied extensions for mixed-mode data management. XDP Application Utilities provide support for system security, optical jukeboxes and application accelerators.

XDP WorkStation software consists of a library of mixed-mode data manipulation routines (XDPLIB) and an extended SQL query processor (XESQL). XDP DataServer and XDP WorkStation software are available in runtime and development versions. Applications in this environment are written in the high-level language "C."

Plexus will release an extended version of Informix incorporating additional features and an Extended Development Environment (XDE) for mixed-mode application development in the second half of 1987. The Plexus XDE will run in the Microsoft Windows environment. These enhancements provide a development platform designed to allow value-added resellers (VARs), original equipment manufacturers (OEMs), application programmers at the customer site, or Plexus to quickly develop mixed-mode applications.

APPLICATIONS AND BENEFITS

Plexus' comprehensive solution for controlling and managing mixed-mode data allows business and government organizations to eliminate data processing frustrations and reduce information management costs.

Companies no longer need to invest in numerous expensive and incompatible hardware and software products to manage diverse modes of information. MIS managers and end users gain significantly greater control over information processing, because Plexus XDP System solutions provide a single system for consolidating disparate modes of information.

The XDP System is ideal for any application managing and integrating large volumes of diverse types of information, including database management, transaction processing, image processing, office automation, and republishing.

For example the system can automate Army tank maintenance and repair records, including handwritten forms, technical documentation with illustrations, and detailed forms cataloguing replacement parts and service calls. In the pharmaceuticals industry, it can automate the paper-intensive process of applying for and gaining Federal Drug Administration approval on new drugs. It can be used by federal and state governments in processing medical insurance forms. And in telecommunications, it is already being used to automate the publishing of U.S. WEST Direct's Yellow Pages business.

Plexus' comprehensive solutions to mixed-mode processing problems also enable VARs, OEMs and customers to identify and develop new business opportunities as they recognize the benefits associated with implementing this new approach to managing information.

In early 1986, Plexus formed a Custom Systems Group (CSG) to begin working directly with VARs, OEMs and large end-user customers in implementing mixed-mode data processing applications. The group is helping these traditional Plexus customers identify and develop new business opportunities made possible by Plexus XDP System applications.

MARKET POTENTIAL

The XDP System advances Plexus from the UNIX super-microcomputer market into the heart of the $50 billion commercial computer systems market. The company's research and independent market studies indicate that the demand for complex data processing solutions which require products like the Plexus XDP System is a significant segment of the commercial data processing market and is growing faster than the market as a whole.

Plexus initially will focus on providing XDP Systems to customers in several of its existing markets. These markets include military, civilian government, pharmaceuticals/health care, and the Regional Bell Operating Companies.

Through its Custom Systems Group (CSG) Plexus has established a strategic partnership program with key customers and third parties in each of these target markets to develop XDP System application programs optimized to solve their unique data processing problems.

XDP SYSTEMS COMPONENTS

XDP Workstation

Additional workstations communicate with each other and over an Ethernet local area network.

DP DataServer

The XDP DataServer is a Plexus P/75 or P/95 with up to 6. 7 gigabytes of magnetic memory and up to eight gigabytes of optical disk storage, or a Plexus P/55 with up to 900 megabytes of magnetic memory.

Optical Disks

The storage requirements for mixed-mode applications typically require optical disk as a mass storage alternative to magnetic storage. Optical drives are offered in the expansion cabinet of the DataServers. A total of eight 12" optical drives are supported in up to 4 expansion cabinets per system. The drives are interfaced to the host through an SCSI adaptor in the host backplane. Applications have the capability to use the optical drive as a write-once file system.

Jukebox

The system includes an optical jukebox for mass storage of images. The maximum capacity of the jukebox is approximately 280 gigabytes with an average access time of 7-10 seconds.

Film and Fiche Scanners

The system is designed to support microfilm and/or microfiche scanners as host peripherals, interfaced into the bus of a Plexus DataServer.

Compression/Decompression/Scaling Co-Processor

Co-processor hardware in the backplane of the AT performs compression, decompression, and scaling of images. CCITT Group 3 and Group 4 algorithms are supported.

Facsimile Support

Direct connection to facsimile machines is supported by use of a facsimile communications card in the backplane of the workstation. Image information may be sent directly over facsimile lines through this card.

Scanners

The workstation can support a range of attached scanning devices for input of images from paper. The first released scanner operates at resolutions of 200, 240, 300, and 400 dots-per-inch, with scan times from three to 20 seconds. Both flatbed and page feed scanners are supported, up to 50 sheets per batch. Sheet sizes of 8.5" x 11" and 8.5" x 14" are supported. The scanners are interfaced to the ATs through an interface card in the AT backplane.

Print Server

The system supports a range of laser printer output devices. Typically, the laser printers will be controlled by PC-AT host that is attached to the network. Images are sent to the print server in compressed form, spooled, and decompressed locally when prepared to print. The PC-AT host is responsible for spooling (i.e. job control and workflow).

OCR Server

Optical Character Recognition functions are provided for both batch and interactive operation. The OCR device is capable of running at a minimum of 60 characters per second with greater than 99% accuracy.

DP DataServer Network Communication

Images and data are carried by a standard Ethernet network. TCP/IP protocols are used for XDP DataServer transfers.

XDP SYSTEM OPERATING ENVIRONMENT

Operating Systems

The XDP DataServer operates under the UNIX Sys 5 operating system. The XDP WorkStation operating system is DOS 3.1.

Database Manager

The system provides extensions to traditional database technology by offering transparent support of text and image data types in a relational database paradigm.

Processing SQL Queries

A key function of the system is the ability to process extended WQL queries extended for mixed-mode data.

Windows

Applications written for the system platform will execute under the Microsoft Windows graphical environment. Windows provides a standard target for development that supports mixed-mode data, and guarantees application compatibility with future operating system enhancements.

Applications

The specific user application code executes on the workstation, generating SQL queries that are sent over the network to the XDP DataServer. This application code may be written by an experienced end user or by Plexus.

Security, File and Record Locking

Data security is primarily available through the standard SQL GRANT and REVOKE access statements. File and record locking rely, for the most part, on facilities provided by the host operating system, the RDSMS, and network software.

TRADEMARKS

Plexus is a registered trademark and XDP and XDP System are trademarks of Plexus Computers, Inc. UNIX is a registered trademark of AT&T IBM PC-AT is a registered trademark of International Business Machines Corporation Ethernet is a registered trademark of Xerox Corporation MS-DOS and Windows are registered trademarks of Microsoft Corporation Informix is a registered trademark of Informix Software, Inc.

For further information, contact:

Julian Day
Microprocessor Applications
(02) 908 3666

or

Lesley Angus
Microprocessor Applications
(03) 894 1500

Awk-ward yacc ::= lex

Bob Buckley

School of Mathematics, Physics, Computing and Electronics,
Macquarie University,
NSW 2109.

ABSTRACT

Some of the tedium of writing simple lexical phases for a *yacc* parser can be eliminated by using *awk* to extract input for *lex* from *yacc* source code. In other cases, a slightly more complex treatment is required. This article describes a small *awk*-script which can save time in the development and testing of parsers and languages.

Introduction.

One of the most boring parts of developing a new translator is the process of producing yet another lexical analyser. If you are using *yacc* to produce a parser, the lexical analyser tends to be particularly tedious because it is relatively straight forward, given the *yacc* input. The maintenance of both the lexical analyser and the parser while you develop a language is tedious.

This short article shows how the amount of work was diminished in one project. The technique should be applicable to other projects.

Yacc input.

Input to *yacc* enumerates lexical items (terminal symbols). It isn't difficult to extract most of the information from a *yacc* file and automatically generate input for *lex*. Terminals are of two forms: they are explicitly listed as

`%term` *terminal_list*

or they appear in the grammar as single character symbols surrounded by quotes. Consider the following *yacc* input:

```
%term IF THEN ELSE
%left '+'
%left '*'
%%
block: '{' stmts '}';
stmts: stmt | stmts stmt ;
stmt: IF exp THEN stmt | IF exp THEN stmt ELSE stmt | block | var '=' exp ;
exp: exp '+' exp | exp '*' exp | '(' exp ')' | var ;
var: 'a' | 'b' | 'c' | 'd' ;
%%
#include "lex.c"
```

A suitable lexical analyser might have the form:

```
%%
[ \t\n]                 ; /* ignore whitespace */
--.*$                   ; /* ignore comments? */
IF                      return(IF);
THEN                    return(THEN);
ELSE                    return(ELSE);
                        return(yytext[0]);/* single character symbols */
```

In some cases, this can be automatically extracted from the *yacc* file using the following *awk*-script.

```
BEGIN                   {
                         print "%%" ;
                         print "[ \\t\\n]\t; /* ignore whitespace* /" ;
                         print "--.*$\t; /* ignore comments? */" ;
                        }
/^%term/                {for(i=2;i<=NF;i++) print $i"\treturn("$i");"; }
END                     {print ".\treturn(yytext[0]);/* single char symbols */"; }
```

This over simple approach has a few problems. One difficulty is in avoiding some special cases of items used as tokens - particularly C reserved words and *lex*/*yacc* macros (eg. *lex* uses BEGIN). The other problem is that symbols (eg. identifiers, numbers, etc.) normally require a more complex treatment. Some terminal symbols have semantics attached.

To provide this flexibility, an extended capability is needed. The following extensions were made to the input language:

```
%lex token pattern
%lex <type> token pattern semantics
```

Awk reads the source and produce both *yacc* and *lex* files. When *awk* reads the above it generates in the *yacc* file, lines of the form:

```
%term token
%term <type> token
```

and lines in the *lex* file of the form:

```
pattern return(token);
pattern { yylval.type = semantics; return(token); }
```

An effort is made to ensure that each line of input to the *awk*-script produces a line of *yacc* source code on the output. The reward for this effort, is reasonable error reporting (unfortunately, *lex* on most systems isn't as good as *yacc* in this respect).

The following *awk*-script will deal with both the simple case above, and this more complex situation.

```
BEGIN                   {
                         lexfile = "prep.l";
                         print "%%" >lexfile;
                         print "[ \\t\\n]|--.*$\t;" >lexfile;
                        }
/^%lex/&&$2~/^</        {
                         print "%term",$2,$3;
                         type = substr($2,2,length($2)-2);
                         print $4 "\t{ yylval." type "="$5"; return("$3");}" >lexfile;
                        }
/^%lex/&&$2~/^[^<]/     {
                         print "%term",$2;
                         print $3 "\treturn(" $2 ");" >lexfile;
                        }
/^%term/                { for(i=2;i<=NF;i++) print $i"\treturn("$i");" >lexfile;}
!/^%lex/                { print $0; }
END                     { print ".\treturn(yytext[0]);" >lexfile; }
```

It is important to remember that the order of *lex* patterns is significant. The order from the source file is preserved in the output.

The following example shows how a more conventional treatment of identifiers can be incorporated using this scheme:

```
%union {
        struct symbol *symb;
}
%term IF THEN ELSE END
%lex BBEGIN BEGIN
%lex <symb> VAR [a-zA-Z][a-zA-Z0-9]* symtab()
%left '+'
%left '*'
%%
block: BBEGIN stmts END;
stmts: stmt | stmts stmt ;
stmt: IF exp THEN stmt | IF exp THEN stmt ELSE stmt | block | VAR '=' exp ;
exp: exp '+' exp | exp '*' exp | '(' exp ')' | VAR ;
%%
struct symbol * symtab();
#include "lex.c"
```

There are some slight problems here. In this example, the description for VAR must follow the others for *lex* to operate as expected. The *awk*-script does not check syntax or report any errors. Care is needed to ensure that fields are acceptable to *awk* (ie. contain no blanks or tabs) or a special mechanisms will be needed.

Interaction with *make*.

In the simple case, the *lex* input depends on the *yacc* file. Assuming the *awk*-script is in the file prep.awk, the following entry in *Makefile* would be adequate:

```
x.o: x.y prep.c
prep.l : x.y
        awk -f prep.awk x.y >/dev/null
```

Make knows how to make prep.c from prep.l. The only problem is that *lex* is used whenever x.y changes. In most cases, this can be avoided with a construct like:

```
x.o : x.y lex.c prep.l
prep.l : x.y
        awk -f prep.awk x.y >/dev/null
        cmp -s prep.l lex.l || cp prep.l lex.l
```

In this case, x.y includes lex.c to access the lexical analyser. This version only invokes *lex* if the lexical analyser has changed, which is generally not the case. The dependancy of x.o on prep.l forces *make* to check that lex.c is current.

In the more complex case, both *lex* and *yacc* input depend on a single file. Any changes to that file will result in a relatively complete rebuild. The following treatment will usually work.

```
x.o : x.y lex.c
x.y : x.yl
        awk -f prep.awk x.yl >x.y
        cmp -s prep.l x.l && rm prep.l || mv prep.l lex.l
lex.l : ; touch lex.l # create a file if it doesn't exist
```

Generally, changes are made to the *yacc* part of the file and don't affect the lexical part. As *lex* is relatively slow, it is only used when the *lex* input changes.

Extensions.

This simple technique is easily extended. Instead of automatically generating white-space and comment patterns, other lexical processing could be kept in the single source file by allowing lines of the form:

```
%ignore pattern
%ignore pattern action
```

When (and if) a translator starts to stabilise, the performance of its lexical analyser may become an issue (code generated by *lex* is not known for its performance). Typically, one produces a hand written 'scanner'. In some cases it will be worth retaining some of the automation used to build the lexical analyser (eg. to build a reserved word table).

Conclusions.

A simple technique is given to reduce some of the tedious activity in using *yacc*. This technique is particularly useful during the early stages of translator development. Production code will probably require a higher performance lexical analyser than *lex* usually produces.

The ability to unify syntax and lexical specification into a single source file was recognised by the authors of *aardvark*/*llama* software as an advantagous. This shows that *lex* and *yacc* can be used in a similar fashion.

This technique does not avoid the need for a reasonable understanding of *lex* and its input.

Acknowledgements.

This technique was developed while working at ANU with Brian Molinari and Chris Johnson. They created the situation in which this technique became appropriate. Chris pointed out the similarity to *aardvark*/*llama*.

Mail Links from VMS to UNIX

Robert Smart <smart@ditmelb.oz>

CSIRO Division of Information Technology
55 Barry St, Carlton, Vic 3053"

The problem of interconnecting the two most common operating systems in the academic and research environment, Unix and VMS, particularly for the purpose of transferring mail, has a number of solutions of varying degrees of complexity, cost and neatness. Some standard Digital products can be used to link VMS and Unix. A common solution is to have an Ultrix VAX which can then talk DECnet mail (MAIL-11 protocol) to VMS machines, and SMTP over TCP/IP to other Unix machines. Another possibility is X.400. X.400 implementations exist for Unix in the form of EAN (or the slightly souped up version sold by the Sydney Development Corp). EAN itself is cheaply available to educational institutions. The latest version is running at munnari, and we have successfully received mail from munnari using Digital's X.400 product. We don't anticipate any great problem establishing 2-way communication when the next release of Digital's X.400 product becomes available.

There are are number of other public domain or cheap third party solutions. The first problem is how to get mail in and out of the VMSmail environment, to send it off to some protocol other than the ones it knows about. The first method discovered was that To: addresses with appended comments (anything following a "!" in a VMSmail address is a comment) would keep the comments. So the user could address mail to something like acsnet!user@domain. This is a horrible hack, and anybody still using it should try to give it up. It doesn't support many ordinary VMSmail functions like reply, or using VMSmail from the command line, or forwarding.

Then Digital introduced products such as PSImail (MAIL-11 protocol over X.25), which used a previously unknown foreign protocol hook. For example for PSImail the user specifies the destination as PSI%dte::username. This was reverse engineered by Kevin Carosso of Hughes Aircraft in the US, and by Phil Taylor of RHBNC London. Carosso's version was the basis of Peter Wishart's excellent ACSNET% interface. It also lead to a more ambitious project, an internet mailer for VMS, namely PMDF.

PMDF (Pascal Memo Distribution Facility) was originally designed by Ira Winston to be a portable and simple system whose main job would be to run CSNET's phonenet protocol, and hence interface to the Unix product mmdfII. The VMS version of PMDF has grown a long way from the portable version, and now supports many protocols, and has a system for choosing routes on the basis of destinations and for altering addresses as required (not as sophisticated as sendmail, but at least comprehensible).

The user interface to PMDF is as clean as it can be with current restrictions in VMSmail (we hope for improvements in version 5 of VMS). The user uses addresses of the form IN%"user@domain". These addresses also appear in the From: field on incoming mail, so reply works correctly, as does forwarding.

PMDF Channels for Interfacing to UNIX

The currently available PMDF channels are most easily interfaced to BSD Unix systems, or at least to systems running sendmail. A uucp interface is in the works, and the guy who is working on it is Kevin Carosso who is very good, so there is a good chance that it will be in the next release. Meanwhile Andrew Worsley, who now works for DIT, is trying to get sendmail up on our System V system (an ICL Clan4). It's not trivial: you certainly can't just say "make SysV". Unfortunately you have to turn off SMTP to avoid getting socket code included, and you thus lose the ability to run smtp over other lower level protocols. If anyone has solved this problem we would be interested.

PMDF interfaces exist for all the VMS implementations of TCP/IP that I know of. TCP/IP used to be a difficult or expensive thing to get for VMS, but it is now very easy. Carnegie-Mellon University distribute the CMU-TEK version of TCP/IP for the cost of distribution. To get the license to fill in, send paper mail (no e-mail accepted) to CMU-TEK IP Software Request, Computing Services, Carnegie Mellon University, 4910 Forbes Ave, Pittsburgh, PA15213-3890. The good news is that you get the source, the bad news is that it's in Bliss (but you get the binaries so it's no problem if you don't want to modify it). Monash University is the only site that I know of to receive their copy at the time of writing.

If you have an Ultrix machine you will find that interfacing to it from PMDF via MAIL-11 is not very satisfactory. Warwick Jackson of Praxa in Melbourne has written a PMDF channel that runs SMTP over DECnet. This should be easy to interface to sendmail on an Ultrix machine, and this is being investigated at Melbourne University. Warwick has also written an SMTP over raw X.25 channel, and a Unix version of this is also being investigated. This certainly holds out hope for all the VMS and Unix machines on Austpac to talk to each other directly.

Finally we come down to reality. The only thing linking your VMS machine to your Unix machine is a length of damp string, or perhaps a modem by which one can call the other. Traditional in Australia is a pseudo-kermit protocol system which is used with the early ACSNET! system which is distributed with ACSnet, and the improved ACSNET% system from Peter Wishart. In this system the Unix system logs in to the VAX which plays a completely passive role. If there is anybody with this system who wants to run PMDF but doesn't want to disturb the Unix end of the current arrangement, then I have an equivalent PMDF channel which I coded but never tested: any suckers want to try it?

However PMDF has its own dialup protocol, plus script facility for logging in to other systems. The protocol is the phonenet protocol used by CSNET in the USA. It was first implemented in the Unix mmdf software. MmdfII is available on the BSD 4.3 tapes, but does not seem to be public domain, nor is it easily available in any other way. Further mmdfII is designed as an alternative to sendmail and wants to duplicate many sendmail functions. What was needed was a more efficient implementation of the phonenet protocol, and one which is easy to interface to sendmail. Andrew Worsley, then at Melbourne University, wrote cmdf to fill this need, and this is used by munnari to talk (over X.25) to relay.cs.net (csnet-relay). We use it at the moment to provide dialup access from our Sun to Praxa's Vax. Cmdf is public domain, and can be obtained from me, or from worsley@ditmela.oz. Those who already have it, and are thinking of interfacing it to PMDF may find it worthwhile to at least get the document describing my experiences getting the link going.

THE SUN NETWORK FILE SYSTEM: Design, Implementation and Experience

Russel Sandberg

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA. 94043
(415) 960-7293

Introduction

The Sun Network File System (NFS™) provides transparent, remote access to filesystems. Unlike many other remote filesystem implementations under UNIX®, NFS is designed to be easily portable to other operating systems and machine architectures. It uses an External Data Representation (XDR) specification to describe protocols in a machine and system independent way. NFS is implemented on top of a Remote Procedure Call package (RPC) to help simplify protocol definition, implementation, and maintenance.

To build NFS into the UNIX kernel in a way that is transparent to applications, we decided to add a new interface to the kernel which separates generic filesystem operations from specific filesystem implementations. The "filesystem interface" consists of two parts: the Virtual File System (VFS) interface defines the operations that can be done on a filesystem, while the virtual node (vnode) interface defines the operations that can be done on a file within that filesystem. This new interface allows us to implement and install new filesystems in much the same way as new device drivers are added to the kernel.

In this paper we discuss the design and implementation of the filesystem interface in the UNIX kernel and the NFS virtual filesystem. We compare NFS to other remote filesystem implementations, and describe some interesting NFS ports that have been done, including the IBM PC™ implementation under MS-DOS™ and the VMS™ server implementation. We also describe the user-level NFS server implementation that allows simple server ports without modification to the underlying operating system. We conclude with some ideas for future enhancements.

In this paper we use the term *server* to refer to a machine that provides resources to the network; a *client* is a machine that accesses resources over the network; a *user* is a person "logged in" at a client; an *application* is a program that executes on a client; and a *workstation* is a client machine that typically supports one user at a time.

Design Goals

NFS was designed to simplify the sharing of filesystem resources in a network of non-homogeneous machines. Our goal was to provide a way of making remote files available to local programs without having to modify, or even relink, those programs. In addition, we wanted remote file access to be comparable in speed to local file access.

The overall design goals of NFS were:

Machine and Operating System Independence
: The protocols used should be independent of UNIX so that an NFS server can supply files to many different types of clients. The protocols should also be simple enough that they can be implemented on low-end machines like the PC.

Crash Recovery
: When clients can mount remote filesystems from many different servers it is very important that clients and servers be able to recover easily from machine crashes and network problems.

Transparent Access
: We want to provide a system that allows programs to access remote files in exactly the same way as local files, without special pathname parsing, libraries, or recompiling. Programs should not need or be able to tell whether a file is remote or local.

UNIX Semantics Maintained on UNIX Client

For transparent access to work on UNIX machines, UNIX filesystem semantics have to be maintained for remote files.

Reasonable Performance

People will not use a remote filesystem if it is no faster than the existing networking utilities, such as *rcp*, even if it is easier to use. Our design goal was to make NFS as fast as a small local disk on a SCSI interface.

Basic Design

The NFS design consists of three major pieces: the protocol, the server side, and the client side.

NFS Protocol

The NFS protocol uses the Sun Remote Procedure Call (RPC) mechanism [1]. For the same reasons that procedure calls simplify programs, RPC helps simplify the definition, organization, and implementation of remote services. The NFS protocol is defined in terms of a set of procedures, their arguments and results, and their effects. Remote procedure calls are synchronous, that is, the client application blocks until the server has completed the call and returned the results. This makes RPC very easy to use and understand because it behaves like a local procedure call.

NFS uses a stateless protocol. The parameters to each procedure call contain all of the information necessary to complete the call, and the server does not keep track of any past requests. This makes crash recovery very easy; when a server crashes, the client resends NFS requests until a response is received, and the server does no crash recovery at all. When a client crashes, no recovery is necessary for either the client or the server.

If state is maintained on the server, on the other hand, recovery is much harder. Both client and server need to reliably detect crashes. The server needs to detect client crashes so that it can discard any state it is holding for the client, and the client must detect server crashes so that it can rebuild the server's state.

A stateless protocol avoids complex crash recovery. If a client just resends requests until a response is received, data will never be lost due to a server crash. In fact, the client cannot tell the difference between a server that has crashed and recovered, and a server that is slow.

Sun's RPC package is designed to be transport independent. New transport protocols, such as ISO and XNS, can be "plugged in" to the RPC implementation without affecting the higher level protocol code (see appendix 3). NFS currently uses the DARPA User Datagram Protocol (UDP) and Internet Protocol (IP) for its transport level. Since UDP is an unreliable datagram protocol, packets can get lost, but because the NFS protocol is stateless and NFS requests are idempotent, the client can recover by retrying the call until the packet gets through.

The most common NFS procedure parameter is a structure called a file handle (fhandle or fh) which is provided by the server and used by the client to reference a file. The fhandle is opaque, that is, the client never looks at the contents of the fhandle, but uses it when operations are done on that file.

An outline of the NFS protocol procedures is given below. For the complete specification see the *Sun Network Filesystem Protocol Specification* [2].

null() returns ()
Do nothing procedure to ping the server and measure round trip time.

lookup(dirfh, name) returns (fh, attr)
Returns a new fhandle and attributes for the named file in a directory.

create(dirfh, name, attr) returns (newfh, attr)
Creates a new file and returns its fhandle and attributes.

remove(dirfh, name) returns (status)
Removes a file from a directory.

getattr(fh) returns (attr)
Returns file attributes. This procedure is like a stat call.

setattr(fh, attr) returns (attr)
Sets the mode, uid, gid, size, access time, and modify time of a file. Setting the size to zero truncates the file.

read(fh, offset, count) returns (attr, data)
Returns up to *count* bytes of data from a file starting *offset* bytes into the file; **read** also returns the attributes of the file.

write(fh, offset, count, data) returns (attr)
Writes *count* bytes of data to a file beginning *offset* bytes from the beginning of the file. Returns the attributes of the file after the **write** takes place.

rename(dirfh, name, tofh, toname) returns (status)
Renames the file *name* in the directory *dirfh*, to *toname* in the directory *tofh*.

link(dirfh, name, tofh, toname) returns (status)
Creates the file *toname* in the directory *tofh*, which is a link to the file *name* in the directory *dirfh*.

symlink(dirfh, name, string) returns (status)
Creates a symbolic link *name* in the directory *dirfh* with value *string*. The server does not interpret the *string* argument in any way, but saves it and makes an association to the new symbolic link file.

readlink(fh) returns (string)
Returns the string that is associated with the symbolic link file.

mkdir(dirfh, name, attr) returns (fh, newattr)
Creates a new directory *name* in the directory *dirfh* and returns the new fhandle and attributes.

rmdir(dirfh, name) returns(status)
Removes the empty directory *name* from the parent directory *dirfh*.

readdir(dirfh, cookie, count) returns(entries)
Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, file id, and an opaque pointer to the next directory entry called a *cookie*. The *cookie* is used in subsequent **readdir** calls to start reading at a specific entry in the directory. A **readdir** call with the *cookie* of zero returns entries starting with the first entry in the directory.

statfs(fh) returns (fsstats)
Returns filesystem information such as block size, number of free blocks, etc.

New fhandles are returned by the **lookup**, **create**, and **mkdir** procedures that also take an fhandle as an argument. The first remote fhandle, for the root of a filesystem, is obtained by the client using the RPC based MOUNT protocol. The MOUNT protocol takes a directory pathname and returns an fhandle if the client has access permission to the filesystem containing that directory. The reason for making this a separate protocol is that this makes it easier to plug in new filesystem access checking methods, and it separates out the operating system dependent aspects of the protocol. Note that the MOUNT protocol is the only place that UNIX pathnames are passed to the server. In other operating system implementations the MOUNT protocol can be replaced without having to change the NFS protocol.

The NFS protocol and RPC are built on top of the Sun External Data Representation (XDR) specification [3]. XDR defines the size, byte order and alignment of basic data types such as string, integer, union, boolean and array. Complex structures can be built from the basic XDR data types. Using XDR not only makes protocols machine and language independent, it also makes them easy to define. The arguments and results of RPC procedures are defined using an XDR data definition language that looks a lot like C declarations. This data definition language can be used as input to an XDR protocol compiler that produces the structures and XDR translation procedures used to interpret RPC protocols [11].

Server Side

Because the NFS server is stateless, when servicing an NFS request it must commit any modified data to stable storage before returning results. The implication for UNIX based servers is that requests that modify the filesystem must flush all modified data to disk before returning from the call. For example, on a **write** request, not only the data block, but any modified indirect blocks and the block containing the inode must be flushed if they have been modified.

Another modification to UNIX necessary for our server implimentation is the addition of a generation number in the inode, and a filesystem id in the superblock. These extra numbers make it possible for the server to use the inode number, inode generation number, and filesystem id together as the fhandle for a file. The inode generation number is necessary because the server may hand out an fhandle with an inode number of a file that is later removed and the inode reused. When the original fhandle comes back, the server must be able to tell that this inode number now refers to a different file. The generation number has to be incremented every time the inode is freed.

Client Side

The Sun implementation of the client side provides an interface to NFS that is transparent to applications. To make transparent access to remote files work we had to use a method of locating remote files that does not change the structure of path names. Some UNIX based remote file access methods use pathnames like *host:path* or */../host/path* to name remote files. This does not allow real transparent access as existing programs that parse pathnames have to be modified.

Rather than doing a "late binding" of file address, we decided to do the hostname lookup and file address binding once per filesystem by allowing the client to attach a remote filesystem to a directory with the *mount* command. This method allows the client to deal with hostnames only once, at mount time. It also allows the server to limit access to filesystems by checking client credentials. The disadvantage is that remote files are not available to the client until a mount is completed.

Transparent access to different types of filesystems mounted on a single machine is provided by a new filesystem interface in the kernel [13]. Each "filesystem type" supports two sets of operations: VFS interface defines the procedures that operate on the filesystem as a whole; and the Virtual Node (vnode) interface defines the procedures that operate on an individual file within that filesystem type. Figure 1 is a schematic diagram of the filesystem interface and how NFS uses it.

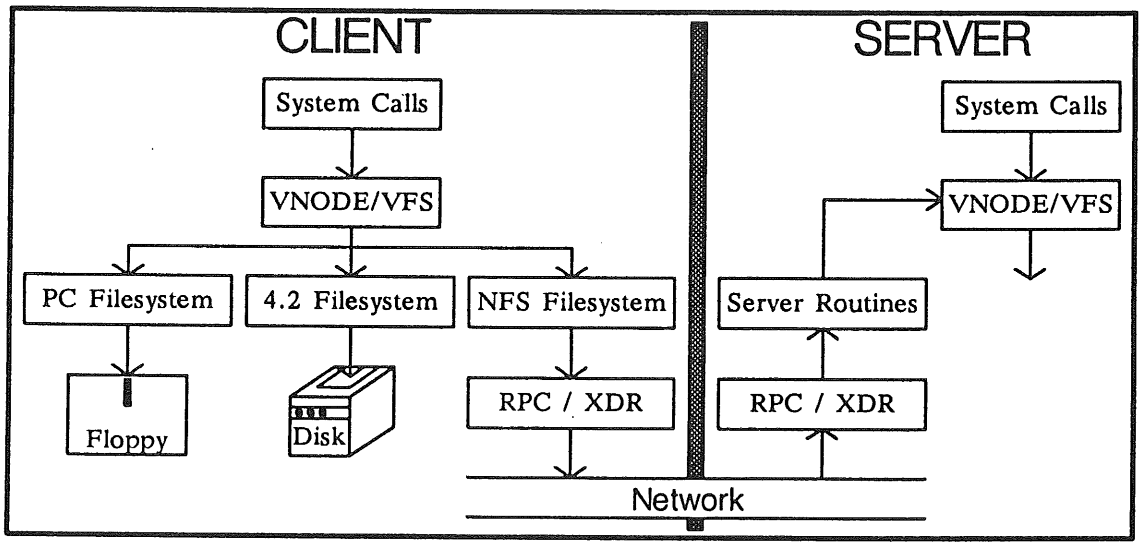


Figure 1

The Virtual File System Interface

The VFS interface is implemented using a structure that contains the operations that can be done on a filesystem. Likewise, the vnode interface is a structure that contains the operations that can be done on a node (file or directory) within a filesystem. There is one VFS structure per mounted filesystem in the kernel and one vnode structure for each active node. Using this abstract data type implementation allows the kernel to treat all filesystems and nodes in the same manner without knowing which underlying filesystem implementation it is using.

Each vnode contains a pointer to its parent VFS and a pointer to a mounted-on VFS. This means that any node in a filesystem tree can be a mount point for another filesystem. A **root** operation is provided in the VFS to return the root vnode of a mounted filesystem. This is used by the pathname traversal routines in the kernel to bridge mount points. The **root** operation is used instead of a pointer so the root vnode for each mounted filesystem can be released. The VFS of a mounted filesystem also contains a pointer back to the vnode on which it is mounted so that pathnames that include ".." can also be traversed across mount points.

In addition to the VFS and vnode operations, each filesystem type must provide **mount** and **mount_root** operations to mount normal and root filesystems. The operations defined for the filesystem interface are given below. In the arguments and results, vp is a pointer to a vnode, dvp is a pointer to a directory vnode and devvp is a pointer to a device vnode.

Filesystem Operations	
mount(varies)	System call to mount filesystem
mount_root( )	Mount filesystem as root
VFS Operations	
unmount(vfs)	Unmount filesystem
root(vfs) returns(vnode)	Return the vnode of the filesystem root
statfs(vfs) returns(statfsbuf)	Return filesystem statistics
sync(vfs)	Flush delayed write blocks
Vnode Operations	
open(vp, flags)	Mark file open
close(vp, flags)	Mark file closed
rdwr(vp, uio, rwflag, flags)	Read or write a file
ioctl(vp, cmd, data, rwflag)	Do I/O control operation
select(vp, rwflag)	Do select
getattr(vp) returns(attr)	Return file attributes
setattr(vp, attr)	Set file attributes
access(vp, mode)	Check access permission
lookup(dvp, name) returns(vp)	Look up file name in a directory
create(dvp, name, attr, excl, mode) returns(vp)	Create a file
remove(dvp, name)	Remove a file name from a directory
link(vp, todvp, toname)	Link to a file
rename(dvp, name, todvp, toname)	Rename a file
mkdir(dvp, name, attr) returns(dvp)	Create a directory
rmdir(dvp, name)	Remove a directory
readdir(dvp) returns(entries)	Read directory entries
symlink(dvp, name, attr, toname)	Create a symbolic link
readlink(vp) returns(data)	Read the value of a symbolic link
fsync(vp)	Flush dirty blocks of a file
inactive(vp)	Mark vnode inactive and do clean up
bmap(vp, blk) returns(devp, mappedblk)	Map block number
strategy(bp)	Read and write filesystem blocks
bread(vp, blockno) returns(buf)	Read a block
brelse(vp, bp)	Release a block buffer

Notice that many of the vnode procedures map one-to-one with NFS protocol procedures, while other, UNIX-dependent procedures such as **open**, **close**, and **ioctl** do not. The **bmap**, **strategy**, **bread**, and **brelse** procedures are used to do reading and writing using the buffer cache.

Pathname traversal is done in the kernel by breaking the path into directory components and doing a **lookup** call through the vnode for each component. At first glance it seems like a waste of time to pass only one component with each call instead of passing the whole path and receiving a target vnode back. The main reason for this is that any component of the path could be a mount point for another filesystem, and the mount information is kept above the vnode implementation level. In the NFS filesystem, passing whole pathnames would force the server to keep track of all of the mount points of its clients in order to determine where to break the pathname; this would violate server statelessness. The inefficiency of looking up one component at a time can be alleviated with a cache of directory vnodes.

Implementation

Implementation of NFS started in March 1984. The first step in the implementation was modification of the 4.2 kernel to include the filesystem interface. By June we had the first "vnode kernel" running. We did some benchmarks to test the amount of overhead added by the extra interface. It turned out that in most cases the difference was not measurable, and in the worst case the kernel had only slowed down by about 2%. Most of the work in adding the new interface was in finding and fixing all of the places in the kernel that used inodes directly, and code that contained implicit knowledge of inodes or disk layout.

Only a few of the filesystem routines in the kernel had to be completely rewritten to use vnodes. *Namei*, the routine that does pathname lookup, was changed to use the vnode **lookup** operation, and cleaned up so that it doesn't use global state. The *direnter* routine, which adds new directory entries (used by **create**, **rename**, etc.), was fixed because it depended on the global state from *namei*. *Direnter* was also modified to do directory locking during directory rename operations because inode locking is no longer available at this level, and vnodes are never locked.

To avoid having a fixed upper limit on the number of active vnode and VFS structures, we added a memory allocator to the kernel so that these and other structures can be allocated and freed dynamically. The memory allocator is also used by the kernel RPC implementation.

A new system call, *getdirentries*, was added to read directory entries from different types of filesystems. The 4.2 *readdir* library routine was modified to use *getdirentries* so programs would not have to be rewritten. However, this change means that programs using *readdir* must be relinked.

Beginning in March 1984, the user level RPC and XDR libraries were ported from the user-level library to the kernel, and we were able to make kernel–to–user and kernel–to–kernel RPC calls in June. We worked on RPC performance for about a month until the round trip time for a kernel to kernel **null** RPC call was 8.8 milliseconds on a Sun-2 (68010). The performance tuning included several speed ups to the UDP and IP code in the kernel.

Once RPC and the vnode kernel were in place the implementation of NFS was simply a matter of writing the XDR routines to do the NFS protocol, implementing an RPC server for the NFS procedures in the kernel, and implementing a filesystem interface which translates vnode operations into NFS remote procedure calls. The first NFS kernel was up and running in mid–August. At this point we had to make some modifications to the vnode interface to allow the NFS server to do synchronous **write** operations. This was necessary since unwritten blocks in the server's buffer cache are part of the "client's state."

Our first implementation of the MOUNT protocol was built into the NFS protocol. It wasn't until later that we broke the MOUNT protocol into a separate, user level RPC service. The MOUNT server is a user level daemon that is started automatically by a mount request. It checks the file /etc/exports which contains a list of exported filesystems and the clients that can import them (see appendix 1). If the client has import permission, the mount daemon does a **getfh** system call to convert the pathname being imported into an fhandle that is returned to the client.

On the client side, the mount command was modified to take additional arguments including a filesystem type and options string. The filesystem type allows one *mount* command to mount any type of filesystem. The options string is used to pass optional flags to the different filesystem types at mount time. For example, NFS allows two flavors of mount, soft and hard. A hard mounted filesystem will retry NFS requests forever if the server goes down, while a soft mount gives up after a while and returns an error. The problem with soft mounts is that most UNIX programs are not very good about checking return status from system calls so you can get some strange behavior when servers go down. A hard mounted filesystem, on the other hand, will never fail due to a server crash; it may cause processes to hang for a while, but data will not be lost.

To allow automatic mounting at boot time and to keep track of currently mounted filesystems, the /etc/fstab and /etc/mtab file formats were changed to use a common ASCII format that is similar to the /etc/fstab format in Berkeley 4.2 with the addition of a type and an options field. The type field is used to specify filesystem type (nfs, 4.2, pc, etc.) and the options field is a comma separated list of option strings, such as rw, hard, and nosuid (see appendix 1).

In addition to the MOUNT server, we have added NFS server daemons. These are user level processes that make an **nfsd** system call into the kernel, and never return. They provide a user context to the kernel NFS server that allows the server to sleep. Similarly, the block I/O daemon, on the client side, is a user level process that lives in the kernel and services asynchronous block I/O requests. Because RPC requests block, a user context is necessary to wait for read-ahead and write-behind requests to complete. These daemons provide a temporary solution to the problem of handling parallel, synchronous requests in the kernel. In the future we hope to use a light-weight process mechanism in the kernel to handle these requests [4].

We started using NFS at Sun in September 1984, and spent the next six months working on performance enhancements and administrative tools to make NFS easier to install and use. One of the advantages of NFS was immediately obvious; the *df* output below is from a diskless machine with access to more than a gigabyte of disk!

```
Filesystem              kbytes    used   avail capacity  Mounted on
/dev/nd0                  7445    5788     912    86%    /
/dev/ndp0                 5691    2798    2323    55%    /pub
panic:/usr               27487   21398    3340    86%    /usr
fiat:/usr/src           345915  220122   91201    71%    /usr/src
panic:/usr/panic        148371  116505   17028    87%    /usr/panic
galaxy:/usr/galaxy        7429    5150    1536    77%    /usr/galaxy
mercury:/usr/mercury    301719  215179   56368    79%    /usr/mercury
opium:/usr/opium        327599   36392  258447    12%    /usr/opium
```

The Hard Issues

Several hard design issues were resolved during the development of NFS. One of the toughest was deciding how we wanted to use NFS. Lots of flexibility can lead to lots of confusion.

Filesystem Naming

Servers export whole filesystems, but clients can mount any sub-directory of a remote filesystem on top of a local filesystem, or on top of another remote filesystem. In fact, a remote filesystem can be mounted more than once, and can even be mounted on another copy of itself! This means that clients can have different "names" for filesystems by mounting them in different places.

To alleviate some of the confusion we use a set of basic mounted filesystems on each machine and then let users add other filesystems on top of that. Remember that this is policy, there is no mechanism in NFS to enforce this. User home directories are mounted on /usr/servername. This may seem like a violation of our goals because hostnames are now part of pathnames but in fact the directories could have been called /usr/1, /usr/2, etc. Using server names is just a convenience. This scheme makes NFS clients look more like timesharing terminals because a user can log in to any machine and the user's home directory will be there. It also makes tilde expansion (where *~username* is expanded to the user's home directory) in the C shell work in a network with many machines.

To avoid the problems of loop detection and dynamic filesystem access checking, servers do not cross mount points on remote **lookup** requests. This means that to see the same filesystem layout as a server, a client has to remote mount each of the server's exported filesystems.

Credentials, Authentication and Security

NFS uses UNIX–style permission checking on the server and client so that UNIX users see very little difference between remote and local files. RPC allows different authentication parameters to be "plugged-in" to the message header so we are able to make NFS use a UNIX flavor authenticator to pass uid, gid, and groups on each call. The server uses the authentication parameters to do permission checking as if the user making the call were doing the operation locally.

The problem with this authentication method is that the mapping from uid and gid to user must be the same on the server and client. This implies a flat uid, gid space over a whole local network. This is not acceptable in the long run and we are working on a network authentication method that allows users to "login" to the network[12]. This will provide a network-wide identity per user regardless of the user's identity on a particular machine. In the mean time, we have developed another RPC-based service called the Yellow Pages (YP) to provide a simple, replicated database lookup service[5]. By letting YP handle /etc/hosts, /etc/passwd and /etc/group we make the flat uid space much easier to administer.

Another issue related to client authentication is super-user access to remote files. It is not clear that the super-user on a machine should have root access to files on a server machine through NFS. To solve this problem the server can map user *root* (uid 0) to user *nobody* (uid −2) before checking access permission. This solves the problem but, unfortunately, causes some strange behavior for users logged in as *root*, since *root* may have fewer access rights to a remote file than a normal user.

Concurrent Access and File Locking

NFS does not support remote file locking. We purposely did not include this as part of the protocol because we could not find a set of file locking facilities that everyone agrees is correct. Instead we have a separate, RPC-based file locking facility. Because file locking is an inherently stateful service, the lock service depends on yet another RPC based service called the status monitor [6]. The status monitor keeps track of the state of the machines on a network so the lock server can free the locked resources of a crashed machine. The status monitor is important to stateful services because it provides a common view of the state of the network.

Related to the problem of file locking is concurrent access to remote files by multiple clients. In the local filesystem, file modifications are locked at the inode level. This prevents two processes writing to the same file from intermixing data on a single write. Because the NFS server maintains no locks between requests, and a write may span several RPC requests, two clients writing to the same remote file can receive intermixed data on long writes.

UNIX Open File Semantics

We tried very hard to make the NFS client obey UNIX filesystem semantics without modifying the server or the protocol. In some cases this was hard to do. For example, UNIX allows removal of open files. A process can open a file, then remove the directory entry for the file so that it has no name anywhere in the filesystem, and still read and write the file. This is a disgusting bit of UNIX trivia and at first we were just not going to support it, but it turns out that all of the programs that we didn't want to have to fix (*csh*, *sendmail*, etc.) use this for temporary files.

What we did to make open file removal work on remote files was check in the client VFS **remove** operation if the file is open, and if so **rename** it instead of removing it. This makes it (sort of) invisible to the client and still allows reading and writing. The client kernel then removes the new name when the vnode becomes inactive. We call this the 3/4 solution because if the client crashes between the **rename** and **remove** a garbage file is left on the server. An entry to *cron* can be added to clean up on the server, but, in practice, this has never been necessary.

Another problem associated with remote, open files is that access permission on the file can change while the file is open. In the local case the access permission is only checked when the file is opened, but in the remote case, permission is checked on every NFS call. This means that if a client program opens a file, then changes the permission bits so that it no longer has read permission, a subsequent **read** request will fail. To get around this problem we save the client credentials in the file table at open time, and use them in later file access requests.

Not all of the UNIX open file semantics have been preserved because interactions between two clients using the same remote file cannot be controlled on a single client. For example, if one client opens a file and another client removes that file, the first client's **read** request will fail even though the file is still open.

Time Skew

Time skew between two clients or a client and a server can cause the times associated with a file to be inconsistent. For example, *ranlib* saves the current time in a library entry, and *ld* checks the modify time of the library against the time saved in the library. When *ranlib* is run on a remote file the modify time comes from the server while the current time that gets saved in the library comes from the client. If the server's time is far ahead of the client's it appears to *ld* that the library is out of date. There were only three programs that we found that were affected by this, *ranlib, ls* and *emacs*, so we fixed them.

Time skew is a potential problem for any program that compares system time to file modification time. We plan to fix this by limiting the time skew between machines with a time synchronization protocol.

Performance

The final hard issue is the one everyone is most interested in, performance. Much of the development time of NFS has been spent in improving performance. Our goal was to make NFS comparable in speed to a small local disk The speed we were interested in is not raw throughput, but how long it takes to do normal work. To track our improvements we used a set of benchmarks that include a small C compile, tbl, nroff, large compile, f77 compile, bubble sort, matrix inversion, make, and pipeline.

To improve the performance of NFS, we implemented the usual read-ahead and write-behind buffer caches on both the client and server sides. We also added caches on the client side for file attributes and directory names. To increase the speed of read and write requests, we increased the maximum size of UDP packets from 2048 bytes to 9000 bytes. We cut down the number of times data is copied by implementing a new XDR type that does XDR translation directly into and out of *mbufs* in the kernel.

With these improvements, a diskless Sun-3 (68020 at 16.67 Mhz.) using a Sun-3 server with a Fujitsu Eagle disk, runs the benchmarks faster than the same Sun-3 with a local Fujitsu 2243AS 84 Mega-byte disk on a SCSI interface.

The two remaining problem areas are **getattr** and **write**. The reason is that *stat*-ing files causes one RPC call to the server for each file. In the local case the inodes for a whole directory end up in the buffer cache and *stat* is just a memory reference. The **write** operation is slow because it is synchronous on the server. Fortunately, the number of **write** calls in normal use is very small (about 5% of all calls to the server, see appendix 2) so it is not noticeable unless the client writes a large remote file.

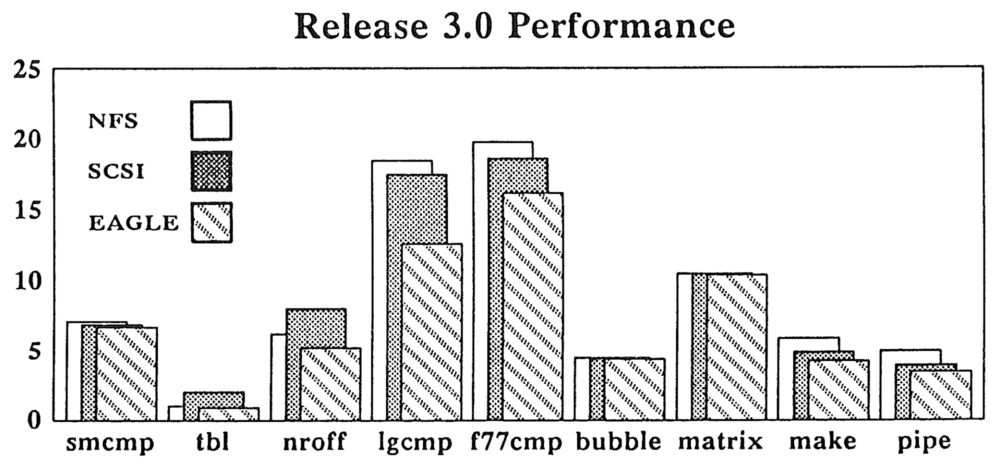


Figure 3

In Figure 3, above, we show some benchmark results comparing NFS and local SCSI disk performance for the current Sun software release. The scale on the left contains unitless numbers. It is provided to make comparison easier.

Many people base performance estimates on raw transfer speed. The current numbers on raw transfer speed are: 250 kilobytes/second for read (cp bigfile /dev/null) and 60 kilobytes/second for write on a Sun-3 with a Sun-3 server.

Other Remote Filesystems

Why, you may ask, do we need NFS when we already have Locus[14], Newcastle Connection[15], RFS[8], IBIS[16] and EFS[10]. In most cases the answer is simple: NFS is designed to handle non-homogeneous machines and operating systems, it is fast, and you can get it today. Other than the Locus system, which provides file replication and crash recovery, the other remote filesystems are very similar to each other.

RFS vs NFS

The AT&T Remote File System (RFS), which has been demonstrated at USENIX and UniForum conferences but is not yet released, will provide much of the same functionality as NFS. It allows clients to mount filesystems from a remote server and access those files in a transparent way. The differences between them mostly stem from the basic design philosophies. NFS provides a general network service, while RFS provides a distributed UNIX filesystem[9]. This difference in philosophy shows up in many different areas of the designs.

Networking

RFS does not use standard network transport protocols, like UDP/IP. Instead it uses a special purpose transport protocol which has not been published, and implementations of it are not generally available. This protocol cannot easily be replaced because RFS depends on properties of the transport virtual circuit to determine when a machine has crashed. NFS uses the RPC layer to hide the underlying protocols, which makes it easy to support different transport protocols without having to change the NFS protocols.

RFS does not use a remote procedure call mechanism, instead it extends the semantics of UNIX system calls so that a system call that accesses a remote file goes over the network and continues execution on the server. When the system call is finished, the results are returned to the client. This protocol is complicated by the fact that both client and server can interrupt a remote system call. In addition, the system calls that deal with filenames had to be modified to handle a partial lookup on the server when a client mount point is encountered in the pathname. In this case the server looks up part of the name then returns control to the client to look up the rest.

Non-Homogeneous Machines and Operating Systems

While NFS currently runs on 25 different vendors hardware, and under Berkeley 4.2, Sun OS, DEC Ultrix, System V.2, VMS and MS-DOS, RFS will run only System V.3-based UNIX systems. The NFS design is based on the assumption that most installations have many different types of machines on their network, and that these machines run widely varying systems. The RFS protocol includes a canonical format for data to help support different machine architectures, but no attempt is made to support operating systems other than System V.3. The NFS design does not try to predict the future. Instead, it includes enough flexibility to support evolving software, hardware, and protocols.

Flexibility

Because RFS is built on proprietary protocols with UNIX semantics built in, it is hard to imagine using those protocols from different operating systems. NFS, on the other hand, provides flexibility through the RPC layer. RPC allows different transport protocols, authentication methods, and server versions to be supported in a single implementation. This allows us, for example, to use an encrypted authentication method for maximum security among workstations, while still allowing access by PC's using a simpler authentication method. It also makes protocol evolution easier since clients and servers can support different versions of the RPC based protocols simultaneously.

RFS uses streams [7] to hide the details of underlying protocols. This should make it easy to plug in new transport protocols. Unfortunately, RFS uses the virtual circuit connection of the transport protocol to detect server and client crashes*. This means that even the reliable byte stream protocol TCP/IP cannot be plugged in because TCP connections do not go away when one end crashes unless there is data flowing at the time of the crash.

Crash Recovery

The RFS uses a stateful protocol. The server must maintain information about the current mount points of all its clients, the open files, directories, and devices held by its clients, as well as the state of all client requests that are in progress. Because it would be very difficult and costly for the client to rebuild the server's state after a server crash, RFS does not do server crash recovery. A server or client crash is detected when the protocol connection fails, at which point all operations in progress to that machine are aborted. When an RFS server crashes it is roughly equivalent, from the client's point of view, to losing a local disk.

Yet if server crashes are rare events, doing no recovery is acceptable. Keep in mind however, that network delays, breaks, or overloading usually cannot be distinguished from a machine crash. As networks grow the possibility of failure increases, and as the connectivity of the network increases so does the chance of a client or server crash. We decided early in the design process that NFS must recover gracefully from machine and network problems. NFS does not need to do crash recovery on the server because the server maintains no state about its clients. Similarly, the client recovers from a server crash simply by resending a request.

Administration

There are two major differences between administration of NFS and RFS. The use of a uid mapping table on RFS servers removes the need for uniform uid to user mapping throughout the network. NFS assumes a uniform uid space and we provide the Yellow Pages service to make distribution and central administration of system databases (like /etc/passwd and /etc/group) easier. NFS also has a MOUNT RPC service for each machine acting as a server. The exported filesystem information is maintained on each machine and made available by this service. RFS uses a centralized name service running on one machine on the network to keep track of advertised filesystems for all servers. A centralized name service was not acceptable in NFS because it forces all clients and servers to use the same protocol for exchanging mount information. By having a separate protocol for the MOUNT service we can support different filesystem access checking and different operating system dependent features of the mount operation.

UNIX Semantics

NFS does not support all of the semantics of UNIX filesystems on the client. Removing an open file, append mode writes, and file locking are not fully implemented by NFS. RFS does implement 100% of the UNIX filesystem semantics. However, if a server crashes or a filesystem is taken out of service, client applications can see error conditions that normally could only happen due to a disk failure. Since this is an error condition that is so severe that it usually means that the whole system has failed, most applications will not even try to recover.

Availability

NFS has been a product for more than a year. Source and support for NFS on Berkeley 4.2 BSD is available through Sun and Mt. Xinu, and for System V.2 through Lachman Associates, The Instruction Set, and Unisoft. RFS has not yet been released.

Conclusion

For a small network of machines all running System V.3, RFS is the obvious choice for remote access to files since it will come with V.3 and it implements all of the UNIX semantics. For a large network or a network of mixed protocols, machine types, and operating systems, NFS is the better choice. It should be understood that NFS and RFS are not mutually exclusive. It will be possible to run both on a single machine.

Porting Experience

In the many ports of NFS to foreign hardware and systems, we have found only a few places where additions to the protocol are helpful. The IBM-PC client side port was done almost

* The exclusive use of transport properties to drive session semantics is a common design flaw in many network applications.

exclusively from the protocol specification, and a simple, user-level server was also implemented from the specification.

NFS has been ported to five different operating systems, two of which are not UNIX-based, and to many different types of machines. Each port had its own interesting problems.

The first port of NFS was to a VAX 750 running Berkeley 4.2 BSD. This was also the easiest port since our code is based on 4.2 UNIX. Modifying the kernel to use the vnode/VFS interface was the most time consuming part of the porting effort. Once the vnode/VFS interface was in, the NFS and RPC code pretty much just dropped in. Some libraries had to be updated, and programs that read directories had to be recompiled. The whole port took about two man-weeks to complete. This port was then used as the distribution source for later ports.

The System V.2 port was done in a joint effort by Lachman Associates and The Instruction Set on a VAX 750. In order to avoid having to port the Berkeley networking code to the System V kernel, an Excelan board was used. The Excelan board handles the Ethernet, IP, and UDP layers. A new RPC transport layer had to be implemented to interface to the Excelan board. Adding the vnode/VFS interface to the System V kernel was the hardest part of the port.

The port to the IBM-PC, done by Geoff Arnold and Kim Kinnear at Sun, was complicated by the need to add a "redirector" layer to MS-DOS to catch system calls and redirect them. An implementation of UDP/IP also had to be added before RPC could be ported. The NFS client-side implementation is written in assembler and occupies about 40K bytes of space. Currently, remote *read* operations are faster than a local hard disk access but remote *write* operations are slower. Overall, performance is about the same for remote and local access.

DEC has ported NFS to Ultrix on a Microvax II. This port was harder than the 4.2 port because the Ultrix release that was used is based on Berkeley 4.3beta. The most time consuming part of the port was, again, installing the vnode/VFS interface. This was complicated by the fact that Berkeley has made many changes to much of the kernel code that deals with inodes.

Another interesting port, while not a different operating system, was the Data General MV 4000 port. The DG machine runs System V.2 with Berkeley 4.2 networking and filesystem added. This made the RPC and vnode/VFS part of the port easy. The hard part was XDR. The MV 4000 has a word addressed architecture, and character pointers are handled very differently than word pointers. There were many places in the code, and especially in the XDR routines that assumed that (char *) == (int *).

As an aid to porting we have implemented a user-level version of the NFS server (UNFS). It uses the standard RPC and XDR libraries and makes system calls to handle remote procedure call requests. The UNFS can be ported to non-UNIX operating systems by changing the system calls and library routines that are used. Our benchmarks show it to be about 80% of the performance of a kernel based NFS server for a single client and server.

The VMS implementation is for the server side only. The basic port was done by Dave Kashtan at SRI. He started with the user-level NFS server and used the EUNICE UNIX-emulation libraries to handle the UNIX system calls. The RPC layer was ported to use a version of the Berkeley networking code that runs under VMS. Some caching was added to the libraries to speed up the system call emulation and to perform the mapping from UNIX permission checking to VMS permission checking.

At the UniForum conference in February 1986, all of the completed NFS ports were demonstrated. There were 16 different vendors and five different operating systems all sharing files over an ethernet.

Also at UniForum, IBM officially announced their RISC-based workstation product, the RT. Before the announcement, NFS had already been ported to the RT under Berkeley 4.2 BSD by Mike Braca at Brown University.

Conclusions

We think that the NFS protocols, along with RPC and XDR, provide the most flexible method of remote file access available today. To encourage others to use NFS, Sun has made public all of the protocols associated with NFS. In addition, we have published the source code for the user level implementation of the RPC and XDR libraries.

Acknowledgements

There were many people throughout Sun who were involved in the NFS development effort. Bob Lyon led the NFS group and helped with protocol issues, Steve Kleiman implemented the filesystem interface in the kernel from Bill Joy's original design. Russel Sandberg ported RPC to the kernel and implemented the NFS virtual filesystem, and Tom Lyon designed the protocol and provided far-sighted inputs into the overall design. David Goldberg worked on many user level programs, Paul Weiss implemented the Yellow Pages, and Dan Walsh is the one to thank for the performance of NFS. The NFS consulting group, headed by Steve Isaac, has done an amazing job of getting NFS out to the world.

References

[1] B. Lyon, "Sun Remote Procedure Call Specification," Sun Microsystems, Inc. Technical Report, (1984).

[2] R. Sandberg, "Sun Network Filesystem Protocol Specification," Sun Microsystems, Inc. Technical Report, (1985).

[3] B. Lyon, "Sun External Data Representation Specification," Sun Microsystems, Inc. Technical Report, (1984).

[4] J. Kepecs, "Lightweight Processes for UNIX Implementation and Applications," USENIX (1985).

[5] P. Weiss, "Yellow Pages Protocol Specification," Sun Microsystems, Inc. Technical Report, (1985).

[6] J. M. Chang, "SunNet," USENIX (1985).

[7] D.L. Presotto and D. M. Ritchie, "Interprocess Communication in the Eighth Edition UNIX System," USENIX Conference Proceedings, (June 1985).

[8] P. J. Weinberger, "The Version 8 Network File System," USENIX Conference Proceedings, (June 1985).

[9] M. J. Hatch, et al., "AT&T's RFS and Sun's NFS, A Comparison of Heterogeneous Distributed File Systems," UNIX World, (December 1985).

[10] C. T. Cole, et al., "An Implementation of an Extended File System for UNIX," USENIX Conference Proceedings, (June 1985).

[11] B. Taylor, "A protocol compiler for RPC," Sun Microsystems, Inc. Technical Report, (December 1985).

[12] B. Taylor, "A Secure Network Authentication Method for RPC," Sun Microsystems, Inc. Technical Report, (November 1985).

[13] S. R. Kleiman, "An Architecture for Multiple File Systems in Sun UNIX," Sun Microsystems, Inc. Technical Report, (October 1985).

[14] Popek, et al., "The LOCUS Distributed Operating System," Operating Systems Review ACM, (October 1983).

[15] D. R. Brownbridge, et al., "The Newcastle Connection or UNIXes of the World Unite!," Software -- Practice and Experience, (1982).

[16] W. F. Tichy, et al., "Towards a Distributed File System," USENIX Conference Proceedings, (June 1985).

Appendix 1

/etc/fstab and /etc/mtab format

The format of the filesystem database files /etc/fstab and /etc/mtab were changed to include type and options fields. The type field specifies which filesystem type this line refers to, and the options field specifies mount and run time options. The options field is a list of comma separated strings. This allows new options to be added, for example when a new filesystem type is created, without having to change the library routines that parse these files. The example below is the /etc/fstab file from a diskless machine.

```
(Filesystem                     mount point       type options)

/dev/nd0                        /                 4.2  rw            1 1
/dev/ndp0                       /pub              4.2  ro            0 0
speed:/usr.MC68010              /usr              nfs  ro,hard       0 0
#opium:/usr/opium               /usr/opium        nfs  rw,hard       0 0
speed:/usr.MC68020/speed        /usr/speed        nfs  rw,hard       0 0
panic:/usr/src                  /usr/src          nfs  rw,soft,bg    0 0
titan:/usr/doctools             /usr/doctools     nfs  ro,soft,bg    0 0
panic:/usr/panic                /usr/panic        nfs  rw,soft,bg    0 0
panic:/usr/games                /usr/games        nfs  ro,soft,bg    0 0
wizard:/arch/4.3alpha           /arch/4.3         nfs  ro,soft,bg    0 0
sun:/usr/spool/news             /usr/spool/news   nfs  ro,soft,bg    0 0
krypton:/usr/release            /usr/release      nfs  ro,soft,bg    0 0
crayon:/usr/man                 /usr/man          nfs  soft,bg       0 0
crayon:/usr/local               /usr/local        nfs  ro,soft,bg    0 0
topaz:/MC68010/db/release       /usr/db           nfs  ro,soft,bg    0 0
eureka:/usr/ileaf               /usr/ops          nfs  soft,bg       0 0
wells:/pe                       /pe               nfs  rsize=1024    0 0
```

Mount Access Permission: the /etc/exports File

The file /etc/exports is used by the server's MOUNT protocol daemon to check client access to filesystems. The format of the file is <filesystem> <access-list>. If the access list is empty the filesystem is exported to everyone. The access-list consists of machine names and netgroups. Netgroups are like mail aliases, a single name refers to a group of machines. The netgroups database is accessed through the Yellow Pages. Below is and example /etc/exports file from a server.

```
(filesystem          access-list)

/usr                 argon krypton
/usr/release
/usr/misc
/usr/local
/usr/krypton         argon krypton phoenix sundae
/usr/3.0/usr/src     systems
/usr/src/pe          pe-users
```

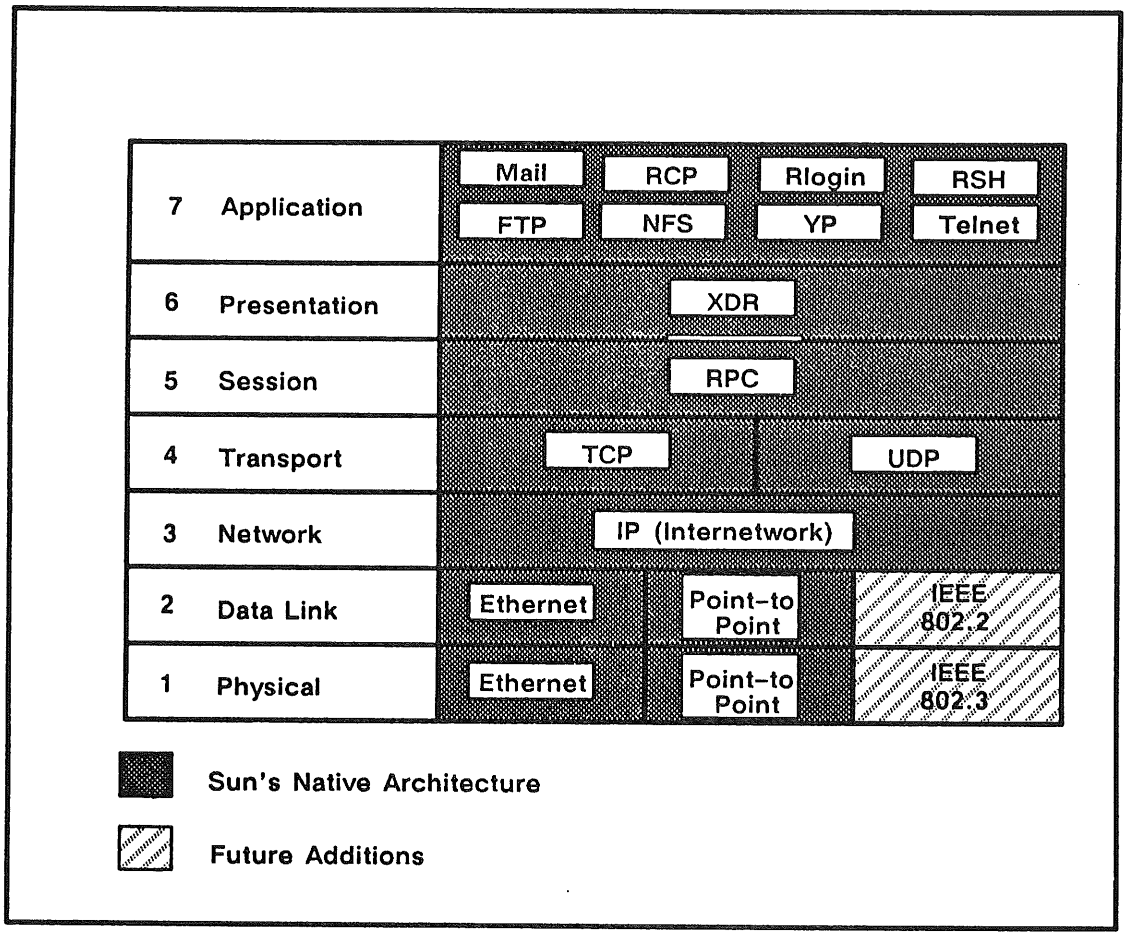
Appendix 2

Below are the server NFS and RPC statistics collected from a typical server at Sun. Statistics are collected automatically each night, using the *nfsstat* command, and sent to a list of system administrators. The statistics are useful for load balancing and detecting network problems. Note that 1499689 calls/day = 62487 calls/hour = 17 calls/second, average over twenty four hours for one server!

```
Server rpc:
calls       badcalls    nullrecv    badlen      xdrcall
1499688     0           0           0           0

Server nfs:
calls       badcalls
1499688     0

null        getattr     setattr     root        lookup      readlink
0  0%       79897   5%  708  0%     0  0%       760709 50%  116712  7%

read        wrcache     write       create      remove      rename
452090 30%  0  0%       50151   3%  25394   1%  5605  0%    687  0%

link        symlink     mkdir       rmdir       readdir     fsstat
683  0%     83  0%      1  0%       1  0%       6960  0%    7  0%
```

Appendix 3

Sun Protocols in the ISO Open Systems Interconnect Model



sun microsystems

Corporate Headquarters
Sun Microsylems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 287815

For U.S. Sales Office locations, call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Sun House
31-41 Pembroke Broadway
Camberley, Surrey GU15 3XD
England
0276 62111
TLX 859017

Germany: (89) 926900-0
UK: 0276 62111
France: (1) 46 30 23 24
Japan: (03) 221-7021

Canadian Headquarters
416 477-6745

Europe, Middle East, and Africa, call European Headquarters:
0276 62111

Elsewhere in the world, call Corporate Headquarters:
415 960-1300
Intercontinental Sales

Shared Libraries in SunOS

Robert A. Gingell
Meng Lee
Xuong T. Dang
Mary S. Weeks

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043

ABSTRACT

The design and implementation of a shared libraries facility for Sun's implementation of the UNIX† operating system (SunOS) is described. Shared libraries extend the resource utilitization benefits obtained from sharing code between processes running the same program to processes running different programs by sharing the libraries common to them.

In this design, shared libraries are viewed as the result of the application of several more basic system mechanisms, specifically

- kernel-supplied facilities for file-mapping and ''copy-on-write'' sharing;
- a revised link editor supporting dynamic binding; and
- compiler and assembler changes to generate position-independent code.

The use of these mechanisms is transparent to applications code and build procedures, and also to library source code written in higher-level languages. Details of the use and operation of the mechanism are provided, together with the policies by which they are applied to create a system with shared libraries. Early experiences and future plans are summarized.

1. Introduction

The UNIX operating system has long achieved efficiencies in memory utilization through sharing a single physical copy of the *text* (code) of a given program among all processes that execute it. However, a program *text* usually contains copies of routines from one or more libraries, and occasionally a program consists *mostly* of library routines. Considering that virtually every program makes use of routines such as *printf*(3), then at any given time there are as many copies of these routines competing for system resources as there are different active programs.

In an environment containing single-user systems, such as workstations, the likelihood of achieving much benefit from sharing multiple copies of entire programs seems small. As the number of programs in a system increases (a guaranteed attribute of each new system release), so does the waste in file storage resources containing yet more copies of common library routines. Thus, there is increasing motivation to extend the benefits of sharing to processes executing *different* programs, by sharing the libraries common to them.

This paper describes the design and implementation of a shared libraries facility for Sun's implementation of the UNIX operating system, SunOS. We discuss our goals for such a facility, our approach

† UNIX is a trademark of Bell Laboratories.

to its design and implementation, and our plans for its use. We also discuss our early experiences, and our plans for the future.

2. Goals

Most of our goals were driven by a desire to have a facility that was as simple to use and evolve as possible. We also wanted to provide mechanisms that were as flexible as possible, so that the work we performed could be used to support other activities and projects. Providing mechanisms with great apparent simplicity would also help motivate their use. To that end, we arrived at the following specific goals:

- **Minimize kernel support.** Clearly, any support we put in the kernel would be very inflexible, and further complicate an already complex environment. We considered an ideal situation to be one involving no kernel changes.
- **Do not require shared libraries.** Although we might make the use of shared libraries the default system behavior, we felt we could not *require* their use or otherwise build fundamental assumptions requiring them into other system components.
- **Minimize new burdens.** The introduction of any new facility creates the potential for new burdens to be imposed upon its users. To minimize these, we decided how shared libraries should impact various groups:
  - **Application programmers:** The use of shared libraries must be transparent to application source code, program build procedures, and the use of standard utilities such as debuggers. It was also considered desirable to be able to use existing object files.
  - **Library programmers:** That a body of library code is to be built as a shared library must also be transparent to its source code. However, it need not be transparent to the procedures used to build the library, and such a goal appeared contradictory in any case – someone has to decide that a shared library will be built. The goal to not change library source was a direct consequence of not having the resources to change the large amount of library code already in existence. Even source alterations such as those used with System V shared libraries [ARNO 86] appeared more than we wished to do.
  - **Administrative:** There should be no requirement to administer and coordinate the allocation of address space. Libraries should be able to evolve and be updated without requiring rebuilding of the programs that used them as long as their interfaces are compatible, and mechanisms would have to be available to handle interface changes.
- **Improve the environment.** Where possible, we wanted our changes to provide functional benefits beyond the resource utilization ones we expected. This included having a great deal of flexibility in easily testing updates to libraries.
- **Performance.** Shared libraries represents a classic time vs. space trade-off opportunity. We were deferring the work of incorporating library code into an address space in order to save both secondary and primary storage space. Thus, we expected to pay a time penalty in programs using shared libraries. However, the expectation was that if sharing of library code really occurred, then the I/O (real) time required to bring in a program and get it executing would be greatly reduced. As long as the CPU time required to merge the program and its libraries did not exceed the I/O time we saved, the apparent performance would be the same or potentially even better. This approach fails if sharing does not occur, or if the system is CPU saturated already.

  Even though a moderate cut in I/O time offers a large window for computation, we felt that an attempt to equal the performance of current systems was unrealistic, and instead set two performance goals permitting a limited degradation in CPU performance for programs that used shared libraries. These goals were:

° ≤ 10% for programs not dominated by start-up costs; and

° ≤ 50% for programs that were dominated by start-up costs.

A program was considered to be dominated by start-up costs if it took less than half a second to execute on a Sun-3/75.

3. Approach

Given our goals for flexibility, the most productive approach was not to build a mechanism *specific* to shared libraries. Rather, by abstracting the general properties we required of shared libraries and providing mechanisms to deliver those properties directly, we hoped to achieve the sought-for benefits and flexibility to address the needs of other projects. The mechanisms we chose were:

- a high degree of memory sharing of general objects (e.g., files) at a fine level of granularity (pages);
- a revised system link editor (*ld*) that supports dynamic loading and binding; and
- compiler changes to generate *position-independent* code (PIC) that need not be relocated for use in different address space arrangements and thus may be directly shared.

3.1. Memory Sharing

The mechanism that provides our memory sharing is a new Virtual Memory (VM) system for SunOS. Although more completely described elsewhere [GING 87], the principal features of the new system include:

- file mapping as its principal mechanism, accessed by programs through the *mmap*(2) system call;
- sharing at the granularity of a file page; and
- a per-page copy-on-write facility to allow run-time modification of a shared object without affecting other users of the object.

The new VM system uses these features internally, so that the act of *exec*'ing a program is reduced to the establishment of copy-on-write mappings to the file containing the program. A "shared library" is added to the address space in exactly the same way, using the general file mapping mechanism. The use of files in this way originated with MULTICS [ORGA 72], and the use of file page mapping to incorporate library support at execution time was established with TENEX [MURP 72] and its evolution as Digital Equipment's TOPS-20. Comparable approaches have been applied with UNIX-based systems as described in [SZNY 86] and [DOWN 84].

3.2. New *ld*

The changes to *ld* reflect an observation that the activities that must occur to execute a program with shared libraries are no different than those to execute one without them, at least conceptually. All that has really changed is when, and over what scope of material, those activities occur. Conceptually, *ld* has been turned into a more general facility available at various times in the life of a program (in perhaps different guises) to perform its link editing function.

The old *ld* built all programs *statically*. Executable (*a.out*) files contained complete programs, including copies of necessary library routines. Executables were created by link editing the program in (usually) a single batch operation using *ld*. *ld* would refuse to build an incomplete executable file.

The new *ld* will build "incomplete" *a.out* files, deferring the incorporation of certain object files until some later time (generally program execution). These deferred link editing operations employ the system's memory management facilities to map to and thus share these objects directly. A "shared library" is simply the code and data constituting a library built as such a *shared object* (*.so*) file. A *.so* is simply one of these "incomplete" *a.out* files that lacks an entry point. It should be noted that a *.so* file can be *any* object, a "library" is simply one of many possible semantic uses for it.

As previously noted, dynamic link editing is still essentially the same operation as static link editing, but occurring at a different time. A link editing operation effects some change to either the material being added, or that to which it is added, or more likely both. However, when an object is changed as the result of such processing, it can no longer be shared with other users of the object, as the change is unlikely to be useful to any program other than the one in which it occurs. Such changes are accommodated automatically by the VM system, using its copy-on-write facilities to create per-process private copies of the pages of the file the process attempts to modify. Thus, the extent of the changes a link edit performs affects the degree to which sharing can occur.

Although a dynamic link edit operation may impact the degree to which sharing can occur in a system, it does not affect the *correctness* of the resulting program. A strong characteristic of our approach is this separation between "right and wrong" vs. "good and bad". Almost any legitimate combination of objects can be link edited into a program at any time (e.g., there are very few "wrong" combinations), but those that maximize sharing will be "best".

3.3. PIC

In the previous section it was observed that code that minimizes the amount of dynamic link editing promotes sharing and is thus "best". To increase the prospects for having the "best" code, we changed our C compiler to optionally generate *position-independent* code (PIC). PIC needs link editing only to relocate references to objects external to the body of code that has been built as PIC, and is thus more sharable. Again, it is not *necessary* to have PIC, just *better*.

However, PIC programs will be slower than non-PIC ones. To localize the link editing for references to global objects, the code refers to such objects indirectly through *linkage tables*. The specific amount of degradation is a function of the number of dynamic references to global objects.

4. Mechanisms

The previous section provided an overview of the approach we have employed, and briefly identified the mechanisms we would use. With this background, we describe the mechanisms in greater detail.

4.1. Compiler Changes

The C compiler has been altered to take a new option (-pic) that causes it to generate PIC. When -pic is specified, the code generated by the compiler changes in the following ways:

- Each function prologue is extended to include the initialization of a register that is used as the base address of a *linkage table* to global objects, this table is called the *global offset table* (*GOT*). For the Motorola 680x0 used in Sun's workstations, this code is:

  ```
  movl #__GLOBAL_OFFSET_TABLE,a5      | Get offset to GOT
  lea pc@(0,a5:L),a5                  | Get absolute address
  ```

  which computes the absolute address of the *GOT* associated with this function based on a PC-relative offset from the function prologue to the table. The register a5 is unavailable for the life of the function, and is one of those that the compiler expects called functions to preserve.

- Each reference to a global data object is generated as a dereference of a pointer in the *GOT*. For example, a reference to the external integer errno in C is generated as:

  ```
  movl a5@(_errno:w),a0               | Get address of _errno
  movl a0@,d0                         | Get contents
  ```

  Currently, the code generation scheme for static data objects is identical to that used for globals. This represents an area for future optimization work.

- Each function call is generated as an assembler pseudo-operation including a "free register", for example:

  ```
  jbsr _foo,a0                        | _foo()
  ```

  for an expression involving a call to the function foo. The assembler will, if _foo is undefined to it, expand the pseudo-operation to an instruction sequence that involves loading a PC-relative reference to an entry in a *procedure linkage table* into the "free register", and then issuing a subroutine call instruction involving the PC in the calculation of the effective address.

The code sequences generated for the 680x0 assume that the linkage tables are of a limited size, specifically no larger than 64K bytes. In the event the tables require a larger size, the compiler can be coerced into generating more clumsy code sequences permitting linkage tables to a full 32 bits in size (by expressing **-pic** as **-PIC**). However, we have yet to find a program that requires the use of this option.

4.2. Assembler Changes

The code generated by the compiler with the **-pic** option requires support from the assembler. The support required is that the assembler generate some new relocation information for certain constructs, and a change in interpretation for some syntactic forms. This support is enabled by the assembler flag **-k**, and is generated automatically by the C compiler driver when invoking the assembler for a compilation that contained the **-pic** or **-PIC** options.

When assembling a module with the **-k** flag enabled, the assembler:

- interprets a relocatable expression in an operand involving an ''immediate'' addressing mode as a PC-relative reference to any symbol involved and generates a PC-relative relocation record for the expression;
- interprets symbolic relocatable expressions in operands involving base-register relative addressing as a reference to the *GOT* entry for the symbol and generates a relocation record indicating such; and
- generates a ''procedure call'' relocation type for all `jbsr` pseudo-operations it assembles.

It should be noted that although examples have been provided using an assembler for the 680x0 processor employed in Sun workstations, the requirements for these special relocation types are architecture independent.

4.3. Link Editor changes

The most extensive changes have been performed to the link editor, *ld*. These not only include changes to the batch form of the link editor (embodied as *ld*), but also the creation of an execution-time version (*ld.so*).

4.3.1. Batch link editor (*ld*)

The batch link editor, *ld*, combines a variety of module types to produce an *a.out* file. How that *a.out* file can be used is very much dependent on what *ld* can determine to do with the information it has been fed. Whereas the previous version of *ld* had to determine *everything* about a program, the new version simply stops working when it runs out of information on the assumption that later events will provide more.

ld's output can be one of two basic types, including:

- a ''simple object'' (*.o* file), produced by simply combining other *.o*'s into a single, larger one (**-r** flag);
- an ''executable'' (*a.out*), which is is either a ''program'' (has an entry point) or a *shared object* (*.so*) (does not have an entry point).

The production of a *.o* file through the use of the **-r** flag is a special use of *ld* that, while useful, is not relevant to the issues being discussed and will not be considered further.

Exactly what gets produced depends on what *ld* was fed in the way of input files and command line options. *ld* will process the following kinds of input files:

- simple object files, *.o*'s;
- *archives*, *.a*'s, conglomerates of simple objects and also referred to as *libraries*; and
- *shared objects*, *.so*'s, also known as dynamically bound executables and sometimes called *shared libraries*.

Each *.o* file is simply concatenated to previous *.o* files in the order it is encountered. In this respect, *ld* is unchanged except that it handles the new relocation operations required by code the assembler generated as PIC.

Each *.a* is searched exactly once as it is encountered – only those entries matching an unresolved external reference are extracted and concatenated. Again, this is exactly as *ld* has always done, with the addition of PIC handling.

Any *.so* encountered is (usually) searched for symbol definitions and references, but does not contribute any material to be concatenated except under certain conditions involving other options (described further below). However, their occurrence in the command line is stored in the resulting *a.out* file and utilized by the execution-time *ld.so* to effect dynamic loading and binding.

ld's **-l** flag is used to specify a short name for an object file to be used as a library. The full name of the object file is derived by adding the prefix *lib* and a suffix of either *.a* or *.so* (for archive or shared library, respectively). The specific suffix applied depends on the binding "mode" *ld* is operating in at the time the **-l** flag is processed. *ld*'s binding "mode" is specified by a new flag, **-B** that takes several keyword arguments:

dynamic	Allow dynamic binding, do not resolve symbolic references, and allow creation of execution-time symbol and relocation information. This is the default setting.
static	Force static binding, implied by options that generate non-sharable executable formats.

-Bdynamic and **-Bstatic** may be specified multiple times and may be used to toggle each other on and off. Like **-l**, their influence is dependent upon their location. When **-Bdynamic** is in effect, any **-l** searches may be satisfied by the first occurrence of either form of library (*.so* or *.a*), but if both are encountered the *.so* form is preferred. Since **-Bdynamic** is the default setting, the use of shared libraries in the construction of a program thus "falls out" from simply installing the *.so* that represents the shared library in the library search path used by *ld*.

If **-Bstatic** is in effect, however, *ld* will refuse to use any *.so* forms of libraries it encounters and continue searching for the *.a* form. Further, an explicit request to load a *.so* file is treated as an error.

After *ld* has processed all its input files and command line options, the form of the output it produces is based on the information it has been able to discern. *ld* first tries to reduce all symbolic references to relative numerical offsets within the executable it is building. To perform this "symbolic reduction", *ld* must know that either

- all information relating to the program has been provided, in particular, no *.so* will be added at execution time; and/or
- this program has an entry point and symbolic reduction can be performed for all symbols having definitions existing in the material it has been provided.

It should be noted that uninitialized "common" areas (essentially all uninitialized C globals) are allocated by the link editor *after* it has collected all references. In particular, this allocation can not occur in a program that still requires the addition of information contained in a *.so* file, as the missing information may affect the allocation process. Initialized "commons", however, are allocated in the executable in which their definition appears.

After *ld* has performed all the symbolic reductions it can, it attempts to transform all relative references to absolute addresses. *ld* is able to do this "relative reduction" only if it has been provided *some* absolute address, either implicitly through the specification of an entry point, or explicitly through other *ld* options. If, after performing all reductions it can, there are no further relocations or definitions to perform, then *ld* has produced a completely linked executable – essentially its old behavior.

However, if any reductions remain, then the executable being produced will require further link editing at execution time in order to be useable. In the data spaces of such executables, *ld* creates an instance of a `link_dynamic` structure that has the label `__DYNAMIC`. The `link_dynamic` structure has the form:

```
struct link_dynamic {
        int     ld_version;             /* Version # */
        struct  link_map *ld_loaded;    /* Loaded objects */
        long    ld_need;                /* Needed objects */
        long    ld_got;                 /* Global offset table */
        long    ld_plt;                 /* Procedure linkage table */
        long    ld_rel;                 /* Relocation table */
        long    ld_hash;                /* Symbol hash table */
        long    ld_stab;                /* Symbol table itself */
        long    (*ld_stab_hash)();      /* Hash function */
        long    ld_buckets;             /* Number of hash buckets */
        long    ld_symbols;             /* Symbol strings */
        long    ld_text;                /* Size of text area */
};
```

This data structure is used by *ld.so* to obtain *.so*'s on which this executable depends, and to find the symbolic and relative reduction operations that remain to be performed. The `link_dynamic` structure contains elements that allow evolution of the interfaces to occur without invalidating existing programs. These include the `ld_version` element, and the incorporation of the hash function for the execution-time symbol table as part of the executable.

4.3.2. Relocation of PIC

As described previously, code generated as PIC contains several new relocation record entries: PC relative, references to entries in a *global offset table* (*GOT*), and references to entries in a *procedure linkage table* (*PLT*).

PC relative relocations are easily handled by *ld*: the value replacing the relocation is simply the offset between the location reference and definition of its target.

GOT and *PLT* entry references are more complex, however. Both of these data structures are allocated by *ld* as part of creating an executable comprised of at least one PIC module, that is, a module containing either *GOT* or *PLT* or both relocation forms. *ld* is responsible for assigning entries in each of these tables for each unique symbol referenced in either a *GOT* or *PLT* reference, and creating a new relocation entry for the table entry. The resulting relocations are then processed just like any other handled by *ld*, by first attempting symbolic and then relative reductions. The table entries themselves are (at least conceptually) indirect pointers to the targets of global references.

4.3.3. *crt0*

Every main program produced by the standard languages is linked with a program prologue module, *crt0*. This module actually contains the program's entry point, and performs various initializations of the environment prior to calling the program's main function or logical starting point. *crt0* was modified to contain a reference to the symbol `__DYNAMIC`. As described above, when *ld* builds an executable requiring execution-time link editing, it defines this symbol as the address of a data structure containing information needed for execution-time link editing operations. If the structure is not needed, any reference to the symbol `__DYNAMIC` is relocated to zero.

Thus, at program start-up, *crt0* tests to see whether or not the program being executed requires further link editing. If not, *crt0* simply proceeds with the execution of the program as it always has – no further processing is involved. However, if `__DYNAMIC` is defined, *crt0* opens the file `/lib/ld.so` and requests the system to map it into the program's address space via the *mmap* system call. It then calls *ld.so*, passing as an argument the address of its program's `__DYNAMIC` structure. *crt0* assumes that *ld.so*'s entry point is the first location in its text. When the call to *ld.so* returns, the link editing operations required to begin the program's execution have been completed.

4.3.4. *ld.so*

After *crt0* transfers control to *ld.so*, *ld.so* executes a short bootstrap routine that performs any relocations *ld.so* itself requires. The process of building *ld.so*, described further below, results in only very simple forms of relocation that can be easily handled by this bootstrap routine. *ld.so* then processes the information contained in the `__DYNAMIC` structure of the program that called it, in order to perform the link editing required to start execution of the program.

ld.so's first action is to examine the `ld_need` entry of the program's `__DYNAMIC` structure. This entry contains an offset relative to the `__DYNAMIC` structure of an array of `link_object` structures. Each element of the array has the structure:

```
struct link_object {
        char    *lo_name;               /* Name of object */
        int     lo_library : 1;         /* Library search */
        short   lo_major;               /* Major version */
        short   lo_minor;               /* Minor version */
};
```

and identifies a *.so* that must be added to the program's address space and link edited. The identification is the name specified on the *ld* command line used to build the program, and includes a bit indicating whether the object was named explicitly or via an *ld* -l option. Some version control information is also recorded, however a discussion of the use of this information is deferred.

For each entry in the `ld_need` array, *ld.so* looks up the file identified and maps it into the process's address space. The location in the address space to which the *.so* is mapped is left to the system to decide, and a given *.so* may reside at different locations in the address spaces of different processes. Failure to find a needed object is a fatal error and results in the program's termination. At the end of the initial program's `ld_need` array, *ld.so* examines the `__DYNAMIC` structure of the first *.so* file it mapped in. It processes that *.so's* `ld_need` array, and proceeds likewise through all the loaded *.so*'s. Any references to already processed *.so* files are ignored.

For each *.so* that is loaded, *ld.so* builds a `link_map` data structure having the form:

```
struct link_map {
        caddr_t lm_addr;                /* Address mapped */
        char    *lm_name;               /* Absolute pathname */
        struct  link_map *lm_next;      /* Next .so */
};
```

Each such structure is placed on a singly linked list in the order it was loaded. The head of the list is rooted in the `ld_loaded` member of the initial program's `__DYNAMIC` structure. This ordering of the loaded *.so*'s is used to establish the search order for undefined symbol look-ups.

After all of the modules comprising the complete program have been placed in the address space, *ld.so* attempts to complete the link editing operations begun by *ld*. Specifically, it attempts to perform first symbolic and then relative reductions on all the references *outside of procedure linkage tables* left in the program. In particular, this includes the allocation of any uninitialized commons (since all information regarding their use is finally present). If all non-procedural references can not be reduced to absolute addresses, then it is because a definition for a given symbol is not available, in which case *ld.so* terminates the program with an ''undefined symbol'' diagnostic.

All non-reduced references in any *PLT*'s in the loaded executables are not processed during program startup. Rather, all such references are initialized to cause the initial calls to the procedures they reference to result in the transfer of control to *ld.so*. Upon receiving control from such a reference, *ld.so* will reduce the original reference to the appropriate absolute address and modify the referencing *PLT* entry to direct future calls directly to the targeted procedure. Deferring the binding of procedure entry points until their first reference saves performing perhaps thousands of unnecessary bindings to entry points programs may never call.

4.3.5. Version Management of *.so*'s

The previous discussion of the handling of *.so* files in the course of processing an *ld* -l option was simplified with respect to *.so* version control. One of the goals of our project was to accommodate the evolution of shared libraries: to permit them to be updated without impacting the programs that used them so long as the interfaces remained compatible.

The *.so* files used as shared libraries actually employ a more complex name than has been described so far, involving a suffix that describes the version of the library contained in the file. Thus, interface version "2" of the C library, in its third compatible revision, would be placed in a *.so* having the name `libc.so.2.3`. The suffix may actually be an arbitrary string of numbers in Dewey-decimal format, although only the first two components are significant to the operation of the link editors at this time.

The first component is called the library's "major version" number, and the second component its "minor version" number. When *ld* records a *.so* used as a library, it also records these two numbers in the database used by *ld.so* at execution time. When *ld.so* finally searches for libraries, it uses these numbers to decide which of multiple versions of a given library is "best", or whether *any* of the available versions are acceptable. The rules it follows are:

- **Major Versions Identical**: the major version used at execution time must exactly match the version found at *ld*-time. Failure to find an instance of the library with a matching major version will cause a diagnostic to be issued and the program's execution terminated.
- **Highest Minor Version**: in the presence of multiple instances of libraries that match the desired major version, *ld.so* will use the highest minor version it finds. However, if the highest minor version found at execution time is less than the version found at *ld*-time, a warning diagnostic will be issued, although execution will continue.

The semantics of version numbers are such that major version numbers should be changed whenever interfaces are changed. Minor versions should be changed to reflect compatible updates to libraries, and programs will silently prefer the highest compatible version they can obtain. If minor version numbers drop, then although the interfaces should remain compatible, it is possible that certain bug fixes or compatible enhancements that the program builder wanted are unavailable: hence the warning diagnostic.

Although the mechanisms for supporting version evolution of shared libraries have been provided, we have not yet provided any tools to automate their use. As before, the detection of incompatibilities remains the responsibility of the library developer.

4.3.6. Link Editor Environment Variables

ld interprets the values of the environment variables LD_LIBRARY_PATH and LD_OPTIONS.

LD_LIBRARY_PATH augments *ld*'s built-in rules for directories to be used when searching for libraries specified with the -l option. If defined, the value of LD_LIBRARY_PATH should be a colon-separated list of directory names (as for the PATH variable of *sh*). The list specified by LD_LIBRARY_PATH is prepended to the list of *ld*'s built-in rules, and follows any further directories specified on the command line with -L options.

LD_OPTIONS specifies a default set of options to *ld*. LD_OPTIONS is interpreted by *ld* just as though its value had been placed on the command line immediately following *ld*'s invocation, as in:

```
% ld $LD_OPTIONS ... other ld arguments ...
```

ld.so also interprets the LD_LIBRARY_PATH environment variable, and may be used to substitute test versions of libraries in their own environments at execution time.

4.3.7. Considerations of Dynamically Linked Programs

Beyond providing a basis for improved sharing of system resources, the ability to defer the binding of library and other code offers a number of other potential advantages in terms of increased flexibility for maintenance and development. However, the environment they create is also inherently more complex, something that the policies governing the application of the mechanisms must address. Some aspects of this more complex environment include:

- **Multiple files**: a dynamically bound program consists not only of the executable file that is the output of *ld*, but also of the files referenced during execution. Moving a dynamically bound program *may* also involve moving a number of other files as well. Moving (or deleting) a file on which a dynamically bound program depends may prevent that program from functioning.

- **Ubiquitous link editor**: the previous behavior of *ld* was to produce only a fully linked executable. Link editing issues could be forgotten or ignored once the executable had been successfully produced. However, deferring some of the link editing means (potentially) deferring some of the errors that could occur. With the new facilities, it is possible for a running program to produce a link editor error.

  Consider the following example: a programmer misspelling in the use of the function call *printf* results instead in a reference to ''pintf''. During testing of the code in which the misspelling occurs, no path to the ''pintf'' reference is ever exercised. However, a later production user does exercise the path. The (no doubt surprised) user will find the program terminated with the message: ''_pintf: undefined''.

  To deal with such problems, *ld* has been provided with an assertion-checking facility that (among other things) can be used to determine if a given program will encounter undefined symbols during execution *if used with the dynamic objects now on the system.* Later erroneous changes to such dynamic objects might still create this problem, however. Program builders wishing to isolate themselves from such problems should simply link their programs statically.

- **Semantic Differences**: there are some semantic differences between the dynamic and static binding algorithms. The differences are not expected to manifest themselves as problems with existing programs, unless such programs engaged in questionable practices in their use of library search ordering. The major semantic difference that can create a problem involves old programs built from several components, where several of those components suddenly become dynamically loadable and others remain static.

  Consider the *ld* command:

  ```
  % ld -o x ... <dc> <sc>
  ```

  The executable `x` consists of several objects including a dynamic component (`<dc>`) and a static component (`<sc>`). `<dc>` was, prior to the introduction of the new mechanisms, an unordered archive file. `<dc>` and `<sc>` both contain definitions for the symbol `bar`. In addition, `<dc>` contains a reference to `bar`. If, in `<dc>`'s prior existence as an unordered static archive, the definition of `bar` preceded its reference, the `ld` operations to build `x` may have satisfied `<dc>`'s reference with the definition from `<sc>`. However, in its dynamic form, `<dc>`'s own definition will be used. This is a consequence of the fact that at execution time, all searches for a symbol definition start with the main program and then all *.so*'s in load order. This behavior preserves the ability to *interpose* on library entry points.

4.4. Debuggers

The debuggers used in the SunOS environment, *adb* and *dbx*, have been modified to deal with the dynamic linking environment provided by the new *ld*. In particular, they understand that symbol definitions may appear after a program starts executing. Such dynamically added symbols are found by noting the creation of the `link_map` structure list in the initial program's `__DYNAMIC` structure, and adding the symbols for the *.so*'s that have been added to the debugger's database of symbols.

Despite our goal for transparency in the tools application programmers use, debugger users must also have some awareness of the use of dynamic linking. For example, if they reference the symbol `printf` in a program that uses a shared C library but has not yet started executing, the debugger will fail to find it. If, however, such a reference has been made after the same program has executed far enough to call the program's `main()`, then `printf` will appear.

5. Policies: Applying the Mechanisms

The previous sections have provided descriptions of our approach to providing a shared library capability through the application of basic mechanisms. We have also described the basic mechanisms involved. In this section, we describe the policies by which we use the mechanisms to build a system that provides and uses shared libraries. In general, the considerations applied in setting these policies were (in decreasing order of priority):

- maximize sharing (resource utilization performance);
- maximize flexibility (enriched environment);
- ''Principle of Least Astonishment'' (user compatibility).

This is to say that a conflict between something that was completely compatible and something that improved sharing or flexibility, generally favored the latter. However, in many cases, it has been possible to accomplish all three considerations.

5.1. System Construction

To meet the goals for resource reduction in the system, the system itself should be built to use shared libraries, and thus, dynamic link editing. This creates the potential for three sorts of problems:

- deferred errors (the so-called ''pintf'' problem) that are manifested after the system is installed;
- the potential for chaos if an important shared library is deleted; and
- the potential for security problems with ''setuid'' programs.

To deal with the problem of deferred errors, a set of programs that are supposed to be self-consistent should be built using the assertion-checking facilities previously described.

To deal with the chaos that would result if (for example) a shared C library were deleted from the system, a number of commands and utilities will not be built with shared libraries. These include but would probably not be limited to: *init*(8), *getty*(8), the shells, *mv*(1), *ln*(1), *ls*(1), *tar*(1) and *restore*(8) – essentially programs that would be necessary to restore the missing library from some other other source.

Finally, programs that are built as ''setuid'' (or ''setgid'' for that matter) are not built to use shared libraries. Such programs could be easily subverted by incorporating a ''trojan horse'' into a library on which they depend.

5.2. Dynamic Binding

To maximize the benefits of shared libraries, we have decided to make their use the default by having the default binding mode for *ld* be **-Bdynamic**. This creates the potential for users of the programs *ld* builds to be ''surprised'' by the special considerations of dynamically linked executables the next time they rebuild their programs. In this case, our preference for maximizing sharing took precedence over the potential for user surprise, a choice we made because we believe:

- most users want the benefits; and
- the mechanisms are sufficiently transparent that the ''potential'' for surprise is not considered to be the same as ''likelihood''.

The greatest impact is expected to be on those users who create programs for shipment to other systems. Such users probably want to be isolated from the various problems that a dynamically linked program can have, and should force their programs to be linked statically. While this may impact existing build procedures, such developers usually take special steps when building production programs (such as removing debugging features and employing extra optimization). The addition of another consideration appeared to be a small cost relative to the benefits obtained by the community through maximizing sharing.

5.3. Use of assertions

To help deal with the potential complexities created by dynamic linking, *ld* has been provided with the ability to validate some assertions about an executable it builds. The assertion checking is invoked with the *ld* flag **-assert**, followed by a keyword argument from one of:

definitions	if the resulting program were run now, there would be no run-time undefined symbol diagnostics;
nosymbolic	there are no symbolic relocation items remaining to be resolved; and
pure-text	the resulting executable requires no further relocations to its text.

Together, these assertions are intended to support the development of production programs by allowing the verification of important properties: for instance that a program will not produce run-time link edit diagnostics, or that a piece of code intended to be a "shared library" is in fact sharable.

5.4. PIC Generation

As has been pointed out, PIC in dynamically linked objects improves their ability to be shared and is thus a more efficient use of system resources. However, PIC executes slower than non-PIC, the degree of degradation being dependent on the number of dynamic indirect references the code incurs. Although refinements to the generated code may ultimately make the performance impact of PIC negligible, we have chosen to make the use of PIC an option. Our expectation is that only code intended to be part of a shared library will be compiled as PIC.

We also expect that few users will enable the generation of PIC in their application programs, simply because it takes extra effort to do so. However, this raises the issue of the binding of non-PIC code to the PIC shared libraries it uses. The binding that must occur involves all references to:

- **commons**: allocated after the program is completely assembled;
- **initialized data**: imported from the shared libraries; and
- **entry points**: supplied by the shared libraries.

The implication of these binding operations is simply that the link editing that implements the binding will render the edited code unsharable.

To improve the degree of sharing for such programs, *ld* can be made to force the allocation of commons and to create aliases for library entry points. These allocations and aliases are created as part of the non-PIC executable, and result in "pure-text" non-PIC programs even if they have dynamically linked components. These options (**-dc** to force the definition of common storage, and **-dp** to force the definition of procedure aliases), are included by the C compiler driver automatically in the *ld* command line it generates.

6. Examples: *.so* Construction

6.1. Shared C Library

The construction of the shared C library involved:

- compilation of all of its C source modules using the **-pic** option;
- modifications of some assembly-language source files so that the assembly source was also position-independent; and
- *ld*'ing the resulting collection of *.o* files to create `libc.so.1.0`.

The modification of the assembly-language source files was, of course, not a requirement for the library to function – simply to make it more sharable. In this area, we fell short of our goal for having shared libraries be transparent to library source code, though happily the amount of assembly source in the system is relatively small.

The *ld* operation, although in reality involving more complex operations resulting from the way we build the various versions of the C library, is conceptually just:

```
% ld -o libc.so.1.0 -assert pure-text *.o
```

assuming the current directory for the command contained only the *.o* files comprising the C library. Since no entry point is supplied, and lacking any other clue to an absolute address, *ld* simply stops processing after it has combined all the object files, built the *GOT* and *PLT*'s, and performed any intra-library PC-relative relocations. The assertion request will cause *ld* to issue a diagnostic if the library requires further relocation to the code contained within it, a sign that a non-PIC object has found its way into the library.

6.2. *ld.so*

The execution-time link-editor, *ld.so* is built with an *ld* command that has the form:

```
% ld -o ld.so -Bsymbolic -assert nosymbolic ... list of modules ...
```

and is conceptually just like other *.so* files. However, it also involves the use of a special binding control option **-Bsymbolic** and the assertion **nosymbolic**.

Normally when *ld* builds a program lacking an entry point or other absolute addressing information, it is unable to perform its symbolic reduction operations simply because it can not assume that symbols from other executable files will not be added later. However, *ld.so* must be self-contained, or else it would require itself to operate and would otherwise pollute the symbol space of the programs it link edits.

The **-Bsymbolic** flag forces *ld* to perform symbolic reduction operations using the information it has now, leaving only relative reductions to be performed – something *ld.so* resolves as part of its bootstrapping operations. The result should be a completely self-contained program, in which all symbolic references are satisfied by its own internal definitions. The **nosymbolic** assertion tests whether or not this is in fact the case.

7. Examples: Application Construction

To illustrate the use of the mechanisms by applications users, we will consider several simple examples of application program construction.

7.1. "Hello World"

The classic simple C program is the one that simply prints "Hello world" on its standard output using *printf* and then exits. In an environment where the standard library path includes a *.so* form of the C library, the command

```
% cc -o hello hello.c
```

generates the *ld* command

```
% ld -e start -dc -dp -o hello /lib/crt0.o hello.o -lc
```

This *ld* command will cause the creation of the executable file `hello`. Since the default behavior of *ld* is to prefer the use of shared libraries, `hello` will be built as an "incomplete" executable requiring the inclusion of the library file `libc.so[.v]` (where `[.v]` represents the required version string) at execution time.

When the program is executed, *crt0* will discover the `__DYNAMIC` structure *ld* left behind and map in the execution-time linker, *ld.so. ld.so* will map in the appropriate version of `libc.so`, allocate any uninitialized commons required by the program, and cause unresolved procedure references in both `hello` and `libc.so` to call *ld.so.* The user's call to *printf* invokes such a call, causing *ld.so* to search first the symbol table of `hello` and then `libc.so` for a definition of *printf.* The definition is found in `libc.so` and the *PLT* entry for the original call is updated to cause future references to go directly to *printf. printf* internally makes other calls to various parts of the C library, each of these intercepted and relocated by *ld.so.*

Although it might be argued that the relocations of intra-C-library calls could have been optimized by prebinding them. However, this would break interposition, as demonstrated by the next example.

7.2. Interposition

Consider the building of the program `hello` again, this time involving a special library, `libinterpose`. This library, like `libc`, is available in a *.so* form. The command used to build `hello` is:

```
% cc -o hello hello.c -linterpose
```

transparently invoking an *ld* command referencing `libinterpose` before `libc`. `libinterpose` defines entry points for various system calls, such as *read* and *write*, that in addition to invoking the required system call also take various statistics on the use of the system calls they surround.

As before, *ld.so* is invoked and maps in the two libraries, first `libinterpose` and then `libc`. The program calls *printf* requiring a relocation to the entry point in `libc`. Eventually, the code that implements *printf* and its descendents issues a call to *write*.

As previously noted, `libinterpose` defines an entry point for *write*. However, so does `libc`, as the standard interface for the *write* system call. *ld.so* resolves the ambiguity by using the ordering it established when mapping in *.so*'s, which places `libinterpose` first. Thus, `libinterpose` is effectively interposed for all uses of *write* in this program. If `hello` itself had defined a *write* entry point, it would have taken precedence over both `libinterpose` and `libc`.

7.3. Mixing Static and Dynamic Binding

Consider a program linked with two shared libraries, `liba` and (automatically) `libc`. A third library, `libb`, however it is only available in an archive, or *.a*, form. These are combined with the program `foo.c` with the command

```
% cc -o foo foo.c -la -lb
```

`foo` references a procedure `bar` defined in both `liba` and `libb`. *ld* handles this problem by recognizing that `liba` contains a definition for `bar`, and ignoring the one provided in `libb`. Thus, even though the material from `liba` is not incorporated into the program until execution time, `libb` is prevented from contributing a definition.

However, suppose `foo` did not reference `bar`, but `liba` did and further, had no definition for it? In this case, *ld* would incorporate the definition from `libb`, and again the intent of the ordering on the command line is followed despite the difference in binding times.

8. Conclusions and Future Work

We have described the design of a shared libraries facility satisfying most of our goals, including:

- no kernel support specific to shared libraries or dynamic linking;
- transparency to application source code and build procedures;
- transparency to library source in higher-level languages; and
- no administrative procedures required to create or use shared libraries.

Some goals for transparency were only partially achieved, the most significant being the potential confusion to those using the system's debugging tools. The need to change some library assembly source is considered an acceptable minor shortcoming.

Although we have only limited experience with the implementation, early performance measurements indicate that we should meet our performance goals for ''average'' programs. These early measurements reveal:

- **PIC degradation**: the use of PIC in libraries does degrade execution time, although in many programs the degradation in negligible. The degradation is most noticeable in those programs that execute primarily in the libraries, and in some cases the degradation fails to meet our limits of 10%. However, we believe there are opportunities for improving the generation of PIC.
- **Start-up costs**: programs previously dominated by start-up costs and that use only a few libraries fall within our 50% goals. We have identified several areas for optimizing the start-up process, including caching the results of library searches. The start-up overhead for

programs that use many libraries is unacceptably large, and is an area we are investigating.

- **Space reduction**: measurements over most of the system's standard utilities suggest that the average per-program savings from the use of shared libraries will be approximately 24K bytes.

During the time we obtained these early measurements, the new VM system on which the work was performed was also being debugged and shaken-out. The measurements were taken in a worst case environment where only the test programs employed shared libraries. Thus, any benefits or problems created by the dynamics of an environment that is based on shared libraries have not been determined, though it is expected that the sharing of the C library will have a positive impact.

Like most technologies, shared libraries and the mechanisms from which we build them can be abused. The execution-time loading we perform clearly has a cost, and excessive use of it in production programs may produce unacceptable performance. However, extensive use during program development adds a new element of flexibility that developers can use to enhance their development environment. An additional consideration is that a library is now a more powerful construct. Previously, the benefits of libraries were in the packaging they provided commonly used facilities. However, that packaging can now be used to provide performance and functional benefits as well.

Our future plans include:

- **Performance enhancement**: continuing efforts in this area for the foreseeable future. An area of particular interest is work to provide different space/time trade-off points than the two provided by the current implementation.
- **Common Link Editor source**: although they can be conceptually viewed as one, at present the two link editors *ld* and *ld.so* are implemented as separate programs. *ld.so* is particularly simplified, a short-cut taken to speed implementation of a first cut at the facility. We would like to build both programs from a common source. Ideally, *ld* should just be an executable jacket to the common code in *ld.so*.
- **Programmatic interface**: some programs, particularly based on interpretive languages such as LISP, can dynamically generate dynamic references. We would like to support the handling of such references through a common mechanism, and thus wish to provide a program-accessible interface to the services now provided invisibly.
- **Different exception handling**: the current disposition of execution-time errors is to abort the program in which they occur. We would like to investigate the program development environments that might be created with other exception handling policies.

9. Acknowledgements

David Goldberg worked on the initial design for a shared libraries prototype that was the basis of this effort. Evan Adams and particularly Richard Tuck spent many hours discussing and helping to refine the philosophies of link editing employed in this design. Bill Joy provided advice and helpful criticism as well as several alternative and interesting implementation strategies. Bill Shannon was helpful in reviewing drafts of this paper.

10. References

[ARNO 86] Arnold, J. Q., "Shared Libraries on UNIX System V", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[DOWN 84] Downing, C. B., F. Farance, "Transparent Implementation of Shared Libraries", *UniForum Conference Proceedings, Washington DC*, /usr/group, January 1984.

[GING 87] Gingell, R. A., J. P. Moran, W. A. Shannon, "Virtual Memory Architecture in SunOS", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[MURP 72] Murphy, D. L., "Storage organization and management in TENEX", *Proceedings of the Fall Joint Computer Conference*, AFIPS, 1972.

[ORGA 72] Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, 1972.

[SZNY 86] Sznyter, E. W., P. Clancy, J. Crossland, "A New Virtual-Memory Implementation for UNIX", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

Virtual Memory Architecture in SunOS

Robert A. Gingell
Joseph P. Moran
William A. Shannon

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043

ABSTRACT

A new virtual memory architecture for the Sun implementation of the UNIX† operating system is described. Our goals included unifying and simplifying the concepts the system used to manage memory, as well as providing an implementation that fit well with the rest of the system. We discuss an architecture suitable for environments that (potentially) consist of systems of heterogeneous hardware and software architectures. The result is a page-based system in which the fundamental notion is that of mapping process addresses to files.

1. Introduction and Motivation

The UNIX operating system has traditionally provided little support for memory sharing between processes, and no support for facilities such as file mapping. For some communities, the lack of such facilities has been a barrier to the adoption of UNIX, or has hampered the development of applications that might have benefited from their availability. Our own desire to provide a shared libraries capability has provided additional incentive for us to explore providing new memory management facilities in the system.

We have also found ourselves faced with having to support a variety of interfaces. These included the partially implemented interfaces we have had in our 4.2BSD-derived kernel [JOY 83] and those specified by AT&T for System V [AT&T 86]. Aggravating these situations were the variations on those interfaces being developed by a number of vendors that were incompatible with or extended the original proposals. Also, entirely new interfaces have been proposed and implemented, most notably in Carnegie-Mellon's MACH [ACCE 86]. There has been no market movement to suggest which, if any, of these would become dominant, and in some cases a specific interface lacked an important capability (such as System V's lack of file mapping).

Finally, our existing implementation is too constraining a base from which to provide the new functionality we wanted. It is targeted to traditional models of UNIX memory management and specifically towards the hardware model of the VAX.‡ The work required to enhance the current implementation appeared to be adding its own new wart to an increasingly baroque implementation, and we were concerned for its long-term maintainability.

Thus, we decided to create a new Virtual Memory (VM) system for Sun's implementation of UNIX, SunOS. This paper describes the architecture of this new system: the goals we had for its design and the constraints under which we operated, the concepts it embodies, the interfaces it offers the UNIX application programmer and its relationship to the rest of the system. Although our primary intent is to

† UNIX is a trademark of Bell Laboratories.
‡ VAX is a trademark of Digital Equipment Corporation

discuss the architectural issues, information relating to the project and its implementation is provided to add context to the presentation.

2. Goals/Non-Goals

Beyond the previously mentioned functional issues of memory sharing and file mapping, our goals for the new architecture were:

- **Unify memory handling.** Our primary architectural goal was to find the general concepts underlying all of the functions we wanted to provide or could envision, and then to provide them as the basis for all VM operations. If successful, we should be able to reimplement existing kernel functions (such as *fork* and *exec*) in terms of these new mechanisms. We also hoped to replace many of the existing memory management schemes in the kernel with facilities provided by the new VM system.
- **Non-kernel implementation of many functions.** If we were successful in identifying and providing the right mechanisms as kernel operations, then it seemed likely that many functions that otherwise would have had to be provided in the kernel could in fact be implemented as library routines. In particular, we wanted to be able to provide capabilities such as shared libraries and the System V interfaces as *applications* of these basic mechanisms.
- **Improved portability.** The existing system was targeted towards a specific machine architecture. In many cases, attributes of this architecture had crept cancerously through the code that implements software-defined functionality. We therefore wanted to describe software-defined objects using data structures appropriate to the software, and relegate machine-dependent code to a lower system layer accessed through a well-defined and narrow interface.
- **Consistent with environment.** We wanted our system to fit well with the UNIX concepts we were not changing. It would not be acceptable to build the world's most wonderful memory management system if it was completely incompatible with the rest of the system and its environment. Particularly important to us in this respect was the use of the file system as the name space for the objects supported by the system. Moreover, we sell systems that are intended to operate in highly networked environments, and thus we could not create a system that presented barriers to the networked environment.

In addition to these *architectural* goals, there were other goals we had for the project as a whole. These *project* goals were:

- **Maintain performance.** Although it is always desirable to tag a project with the label "improves performance", we chose the apparently more conservative goal of simply providing more functionality for the same cost in terms of overall system performance. While the new functionality might enable increased *application* performance, the performance of the system itself seemed uncertain. Further, when one considers that we replaced a mature implementation with one which has not been subjected to several years of tuning, getting back to current performance levels appeared to be an ambitious goal, something later experience has proven correct.
- **Engineer for the future.** We wanted to build an implementation that would be amenable to anticipated future requirements, such as kernel support for "lightweight" processes [KEPE 85] and multiprocessors.

When engaging in a large project, it is often as important to know what one's goals are *not*. In the architectural arena, our principal "non-goals" were:

- **New external interfaces.** As previously noted, a large number of groups were already working on the refinement and definition of interfaces. To the extent possible, we wanted to use such interfaces as had already been defined by others, and to provide those that were sufficiently defined to be implementable and that the market was demanding.
- **Compatible internal interfaces.** An unfortunate characteristic of UNIX is the existence of programs that have some understanding of the system's internals and use this information to

rummage through the kernel by reading the memory device. The changes to the system we contemplated clearly made it impossible for us to try to support these programs, and thus we decided not to fool ourselves into trying.

Relevant project non-goals included:

- **Pageable kernel.** We did not intend to produce an implementation in which the kernel itself was paged – beyond a general desire in principle for the kernel to use less physical memory, we would have satisfied no specific functional goal by having the kernel pageable. However, it has turned out that a considerable portion of the memory that was previously ''wired down'' for kernel use is in fact now paged, although kernel code remains physically locked.
- **Massive policy changes.** Our interests lay in changing the mechanisms and what they provide, not in the policies by which they were administered. Although we would eventually like to support an integrated view of process and memory scheduling using techniques such as working set page replacement policies and balance set scheduling, we decided to defer these to future efforts.

3. Constraints

Working within the framework of an existing system imposed a number of constraints on what we could do. The constraints were not always limits on our flexibility; in fact, those reflecting specific customer requirements provided data that guided us through a number of design decisions. A major constraint was that of compatibility with previous versions of the system – ultimately, compatibility drove many decisions.

One such decision was that the new system would execute existing *a.out* files. This was necessary to preserve the utility of the programs already in use by customers and third parties. An important implication is that the system must provide a binary-compatible interface for existing programs, which means that existing system calls that perform memory management functions must continue to work. In our case, this meant supporting our partial implementation of the 4.2BSD *mmap*(2) system call, which we used to map devices, such as frame buffers, into a process's address space.

Although the system had to be binary-compatible, we did not feel constrained to leave it source-compatible, nor to use *mmap* as the principal interface to the memory management facilities of the system. Users with programs that used interfaces we changed in this manner would have to change their programs the next time they compiled them, but they would not be forced to recompile just to install and continue operating on the new system.

A wide variety of customer requirements implied that the interfaces we would offer would have to present very few constraints on a process's use of its address space. Some applications wanted to manage their address space completely, including the ability to assign process addresses for objects and to use a large, sparsely populated address space. Our own desire to build a base on which many different interfaces could be easily constructed suggested that we wanted as much flexibility as possible in user level address space management. However, other factors and requirements suggested that the system should also be able to control many details of an address space. One such factor was the introduction of a virtual address cache in the Sun-3/200 family of processors, where system control of address assignment would have a beneficial impact on performance. We also wanted to use copy-on-write techniques to enhance the level of sharing in the system, and to do this efficiently required page-level protection.

4. New Architecture: General Concepts

This section describes in general terms the abstractions and properties of the new VM system, and some reflections on the decisions that led to their creation. In many cases, our decisions were not based on obvious considerations, but rather ''fell out'' of a large number of small issues. Although this makes the decisions more difficult to explain, the process by which they were reached increased our confidence that, given our goals and constraints, we had in fact reached the best conclusion.

4.1. Pages vs. Segments

Our earliest decision was that the basic kernel facilities would operate on pages, rather than segments. The major factors in this decision included:

- compatibility with current systems (the 4.2BSD *mmap* is page-based);
- implementing efficient copy-on-write facilities required maintenance of per-page information anyway;
- pages appeared to offer the greatest opportunity to satisfy customer requirements for flexibility; and
- segments could be built as an abstraction on top of the page-based interface by library routines.

The major advantage to a segment-based mechanism appeared simply to be that it was a "better" programming abstraction. Since we could still build abstraction from the page-based mechanisms, and in fact gained some flexibility in building different forms of the abstraction as libraries, providing segments through the kernel appeared to offer little benefit and possibly even presented barriers to accomplishing some of our goals.

Although we believed we could gain the architectural advantages of segments through library routines built on our page-based system, another potential advantage to a segment-based system was the opportunity to implement a compact representation for a sparsely populated address space. However, since we needed per-page information to implement per-page copy-on-write and perform other physical storage management, at the very least we would end up with a mix of page- and segment-oriented data structures. We recognized that we could keep the major implementation advantage of a segment-based system, i.e., the concise description of the mapping for a range of addresses, by viewing it as an optimization (a sort of run-length encoding) of the per-page data structure (a similar scheme is used in MACH.)

4.2. Virtual Memory, Address Spaces, and Mapping

The system's *virtual memory* consists of all its available physical memory resources. Examples include file systems (both local and remote), pools of unnamed memory (also known as *private* or *anonymous* storage, and implemented by the processor's primary memory and *swap space*), and other random access memory devices. Named objects in the virtual memory are referenced through the UNIX file system. This does not imply that all file system objects are in the virtual memory, but simply that all named objects in the virtual memory are named in the file system. One of the strengths of UNIX has been the use of a single name-space for system objects, and we wished to build upon that strength. Some objects in the virtual memory, such as process private memory and our implementation of System V shared memory segments, do not have names. Although the most common form of object is the UNIX "regular file", previous work on SunOS has allowed for many different implementations of objects, which the system manipulates as an abstraction of the original UNIX *inode*, called a *vnode* [KLEI 86].

A process's *address space* is defined by mappings onto the address spaces of one or more objects in the system's virtual memory. As previously discussed, the system provides a page-based interface, and thus each mapping is constrained to be sized and aligned with the page boundaries defined by the system on which the process is executing. Each page may be mapped (or not) independently, and thus the programmer may treat an address space as a simple vector of pages. It should be noted that the only valid process address is one which is mapped to some object, and in particular there is no memory associated with the process itself – all memory is represented by virtual memory objects.

Each object in the virtual memory has an *object address space* defined by some physical storage, the specific form being object-specific. A reference to an object address accesses the physical storage that implements the address within the object. The virtual memory's associated physical storage is thus accessed by transforming process addresses to object addresses, and then to the physical store. The system's VM management facilities may interpose one or more layers of logical caching on top of the actual physical storage used to implement an object, a fact that has implications for *coherency*, discussed below.

A given process page may map to only one object, although a given object address may be the subject of many process mappings. The amount of the object's address space covered by a mapping is an integral multiple of the page size as seen by the process performing the mapping. An important characteristic of a mapping is that the object to which the mapping is made is not required to be affected by the mere *existence* of the mapping. The implications of this are that it cannot, in general, be expected than an object has an ''awareness'' of having been mapped, or of which portions of its address space are accessed by mappings; in particular, the notion of a ''page'' is not a property of the object. Establishing a mapping to an object simply provides the *potential* for a process to access or change the object's contents.

The establishment of mappings provides an *access method* that renders an object directly addressable by a process. Applications may find it advantageous to access the storage resources they use directly rather than indirectly through *read* and *write*. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (e.g., updates changed to a single step rather than a *read*, modify buffer, *write* cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

It is important to note that this *access method* view of the VM system does not directly provide sharing. Thus, although our motivations included providing shared memory, we have actually only provided the mechanisms for applications to *build* such sharing. For the system to provide not only an access method but also the *semantics* for such access is not only difficult or impossible, it is not clear that it is the correct thing to do in a highly heterogeneous environment. However, useful forms of sharing can be built in such environments, as the previous mechanisms for sharing in the kernel (such as the shared program text and file data buffer cache) have been subsumed by kernel programming building on top of these mechanisms.

4.3. Networking, Heterogeneity, and Coherence

Many of the factors that drove our adoption of the access method view of a VM system originated from our goal of providing facilities that ''fit'' with their expected environment. A major characteristic of our environment is the extensive use of networking to access file systems that would be part of the system's virtual memory. These networks are not constrained to consist of similar hardware or a common operating system; in fact, the opposite is encouraged. Making extensive assumptions about the properties of objects or their access creates potentially extensive barriers to accommodating heterogeneity. These properties include such system variables as page sizes and the ability of an object to synchronize its uses. While a given set of processes may *apply* a set of mechanisms to establish and maintain various properties of objects, a given operating system should not *impose* them on the rest of the network.

As it stands, the access method view of a virtual memory maintains the potential for a given object (say a text file) to be mapped by systems running our memory management system but also accessed by systems for which the notion of a virtual memory or storage management techniques such as paging would be totally foreign, such as PC-DOS. Such systems could continue to share access to the object, each using and providing its programs with the access method appropriate to that system. The alternative would be to prohibit access to the object by less capable systems, an alternative we find unacceptable.

A new consideration arises when applications use an object as a communications channel, or otherwise attempt to access it simultaneously. In addition to providing the mapping functions described previously, the VM management facilities also manage a storage hierarchy in which the processor's primary memory is often used as a cache for data from the virtual memory. Since the system cannot assume either that the object will coordinate accesses to it, nor that other systems will in fact cooperate with such coordination, it does not attempt on its own to synchronize the ''virtual memory cache'' it maintains. This is not to say that such objects can not exist, nor that systems will not cooperate; simply that *in general* the system can not make such an assumption. Even within a single system, the sharing that results is a consequence of the system's attempt to use its cache resources efficiently, not part of its defined functionality.

However, the lack of cache synchronization is not the limitation it might first appear. Applications that intend to share an object must employ a synchronization mechanism around their access and this requirement is independent of the access method they use. The scope and nature of the mechanism employed is best left to the application to decide. While today applications sharing a file object must access and update it indirectly using *read* and *write*, they must coordinate their access using semaphores or file locking or some application-specific protocol. In such environments, either caching is totally disabled (resulting in performance limitations) or the applications must employ a function such as *fsync* to ensure that the object is updated. Coherency of shared objects is not a new issue, and the introduction of a new access method simply exposes a new manifestation of an old problem. All that is required in an environment where mapping replaces *read* and *write* as the access method is that an operation comparable to *fsync* be provided.

Thus, the nature and scope of synchronization over shared objects is something that is application-defined from the outset. If the system attempted to impose any automatic semantics for sharing, it might prohibit other useful forms of mapped access that have nothing whatsoever to do with communication or sharing. By providing the mechanism to support coherency, and leaving it to cooperating applications to apply the mechanism, our design meets the needs of applications without providing barriers to heterogeneity. Note that this design does not prohibit the creation of libraries that provide coherent abstractions for common application needs. Not all abstractions on which an application builds need be supplied by the "operating system".

4.4. Historical Acknowledgements

Many of the concepts we have described are not new. MULTICS [ORGA 72] supported the notion of file/process memory integration that is fundamental to our system. TENEX [BOBR 72] [MURP 72] supported a page-based environment together with the notion of a process page map independent of the object being mapped.

5. External Interfaces: System Calls

The applications programmer gains access to the facilities of the new VM system through several sets of system calls. At present, we have defined our principal interface to be a refinement of those provided with 4.2BSD. We also provide interfaces for System V's shared memory operations. The new system also impacted other system calls and facilities. These are described further below. Although these represent the initial interfaces we intend to support, others may be provided in the future in response to market demand.

5.1. 4.2BSD-based Interfaces

The 4.2BSD UNIX specification [JOY 83] included the definition of a number of system calls for mapping files, although the system did not implement them. Earlier releases of SunOS included partial implementations of these calls to support mapping devices such as frame buffers into a process's address space. The basic concepts embedded in the interface were very close to our own, namely a page-based system providing mappings from process addresses to objects identified with file descriptors, and thus working from this base was a natural thing to do.

However, we had problems with the 4.2BSD interfaces due to their sketchy definition. Although the intent was well understood, the lack of an implementation left many semantic issues unresolved or ambiguous. We required some facilities that were not part of the specification, and other facilities were part of the specification but seemed superfluous. Thus, although we did manage to avoid creating an entirely new interface, we did find ourselves refining an existing, but unimplemented one. The process of refinement involved many people; in fact most were external to Sun and involved exchanges utilizing a "VM interest" mailing list supported and maintained by the developers at UC Berkeley, CSRG. Table 1 summarizes our refined interface, and the following sections expand on various areas of refinements.

Table 1 – Refined 4.2BSD Interfaces	
Call	**Function**

`madvise(addr, len, behav)`
`  caddr_t addr; int len, behav;`
	Gives advice about the handling of memory over a range of addresses.

`mincore(addr, len, vec)`
`  caddr_t addr; int len; result char *vec;`
	Determines residency of memory pages. (Will be replaced by more general map reading function.)

`caddr_t`
`mmap(addr, len, prot, flags, fd, off)`
`  caddr_t addr; int len, prot, flags, fd;`
`  off_t off;`
	Establish mapping from address space to object named by `fd`.

`mprotect(addr, len, prot)`
`  caddr_t addr; int len, prot;`
	Change protection on mapped pages.

`msync(addr, len, flags)`
`  caddr_t addr; int len, flags;`
	Synchronizes and/or invalidates cache of mapped data.

`munmap(addr, len)`
`  caddr_t addr; int len;`
	Removes mapping of address range.

5.1.1. *mmap*

The *mmap*(2) system call is used to establish mappings from a process's address space to an object. Its definition is:

caddr_t mmap(addr, len, prot, flags, fd, off)

mmap establishes a mapping between the process's address space at an address *paddr* for *len* bytes to the object specified by *fd* at offset *off* for *len* bytes. The value of *paddr* is an implementation-dependent function of the parameter *addr* and values of *flags*, further described below. A successful *mmap* call returns *paddr* as its result. The address ranges covered by [*paddr*, *paddr* + *len*) and [*off*, *off* + *len*) must be legitimate for the address space of a process and the object in question, respectively. The mapping established by *mmap* replaces any previous mappings for the process's pages in the range [*paddr*, *paddr* + *len*).

The parameter *prot* determines whether *read*, *execute*, *write* or some combination of accesses are permitted to the pages being mapped. The values desired are expressed by or'ing the flags values PROT_READ, PROT_EXECUTE, and PROT_WRITE. It is not expected that all implementations literally provide all possible combinations. PROT_WRITE is often implemented as PROT_READ|PROT_WRITE, and PROT_EXECUTE as PROT_READ|PROT_EXECUTE. However, no implementation will permit a write to succeed where PROT_WRITE has not been set. The behavior of PROT_WRITE can be influenced by setting MAP_PRIVATE in the *flags* parameter.

The parameter *flags* provides other information about the handling of the pages being mapped. The options are defined by a field describing an enumeration of the "type" of the mapping, and a bit-field specifying other options. The enumeration currently defines two values, MAP_SHARED and MAP_PRIVATE. The bit-field values are MAP_FIXED and MAP_RENAME. The "type" value chosen determines whether stores to the mapped addresses are actually propagated to the object being mapped (MAP_SHARED) or directed to a copy of the object (MAP_PRIVATE). If the latter is specified, the initial write reference to a page will create a private copy of the page of the object and redirect the mapping to the copy. The mapping type is retained across a *fork*(2). The mapping "type" only affects the disposition of stores by *this* process – there is no insulation from changes made by other processes. If an application desires such insulation, it should use the *read* system call to make a copy of the data it wishes to keep protected.

MAP_FIXED informs the system that the value of *paddr* must be *addr*, exactly. The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the most effective use of system resources.

When MAP_FIXED is not set, the system uses *addr* as a hint in an implementation-defined manner to arrive at *paddr*. The *paddr* so chosen will be an area of the address space that the system deems suitable for a mapping of *len* bytes to the specified object. All implementations interpret an *addr* value of zero as granting the system complete freedom in selecting *paddr*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the system selects a value for *paddr*, it will never place a mapping at address 0, nor will it replace any extant mapping, nor map into areas considered part of the potential data or stack ''segments''. In the current SunOS implementation, the system strives to choose alignments for mappings that maximize the performance of systems with a virtual address cache.

MAP_RENAME causes the pages currently mapped in the range [*paddr, paddr* + *len*) to be effectively renamed to be the object addresses in the range [*off, off* + *len*). The currently mapped pages must be mapped as MAP_PRIVATE. MAP_RENAME implies a MAP_FIXED interpretation of *addr*. *fd* must be open for write. MAP_RENAME affects the size of the memory object referenced by *fd*: the size is max(*off* + *len* - *1*, *flen*) (where *flen* was the previous length of the object). After the pages are renamed, a mapping to them is reestablished with the parameters as specified in the renaming *mmap*.

The addition of MAP_FIXED and corresponding changes in the default interpretation of *addr* and *mmap*'s return value represent the principal change made to the original 4.2BSD specification. The change was made to remove the burden of managing a process's address space from applications that did not wish it.

5.1.2. Additions

We added one new system call, *msync*. *msync* has the interface

msync(addr, len, flags)

msync causes all modified copies of pages over the range [*addr, addr* + *len*) in system caches to be flushed to the objects mapped by those addresses. *msync* optionally invalidates such cache entries so that further references to the pages will cause the system to obtain them from their permanent storage locations. The *flags* argument provides a bit-field of values which influences *msync*'s behavior. The bit names and their interpretations are:

`MS_ASYNC`	Return immediately
`MS_INVALIDATE`	Invalidate caches

MS_ASYNC causes *msync* to return immediately once all I/O operations are scheduled; normally, *msync* will not return until all I/O operations are complete. MS_INVALIDATE causes all cached copies of data from mapped objects to be invalidated, requiring them to be re-obtained from the object's storage upon the next reference.

5.1.3. Unchanged Interfaces

Two 4.2BSD calls were implemented without change. They were *mprotect* for changing the protection values of mapped pages, and *munmap* for removing a mapping.

5.1.4. Removed: *mremap*

We deleted one system call, *mremap*. Upon reading the 4.2BSD specification, we had the impression that *mremap* was the mapping equivalent of the UNIX *mv* command. However, discussions with those involved in its original specification created confusion as to whether it was in fact supposed to be the equivalent of *mv*, *cp*, or *ln*. In the presence of the uncertainty and lacking any other motivation to include it, *mremap* was dropped from the system.

5.1.5. Open Issues

Two 4.2BSD system calls, *madvise* and *mincore*, remain unspecified. *madvise* is intended to provide information to the system to influence its management policies. Since a major rework of such policies was deferred to a future release, we decided to defer full specification and implementation of *madvise* until that time.

mincore was specified to return the residency status of a group of pages. Although the intent was clear, we felt that a more comprehensive interface for obtaining the status of a mapping was required. However, at present, this revised interface has not been defined.

Also unspecified is an interface for locking pages in memory. We envision either a new *mlock* system call, or a variation on *madvise*.

5.2. System V Shared Memory

The ''System V Interface Definition'' [AT&T 86] defines a number of operations on entities called ''shared memory segments''. Early in our project, we had hoped to implement these operations not as system calls but rather as library routines which built the System V abstractions out of the basic mechanisms supplied by the kernel. Unfortunately, System V shared memory is almost, but not completely the same as, a UNIX file. The primary differences are:

- **name space**: a shared memory segment exists in a name space different from that of the traditional UNIX file system; and
- **ownership and access**: a shared memory segment separates the notion of ''creator'' from ''owner''.

Together, these differences motivated a kernel-based implementation to allocate and manage the different name space (which shared implementation with other System V-specific objects such as semaphores), and to administer the different ownership and access control operations.

Although the databases peculiar to these differences are maintained inside the kernel, the implementation of the objects and access are built from the standard notions. Specifically, the memory object representing the shared memory segment exists as an unnamed object in the system's virtual memory, and the operation which attaches processes to it performs the internal equivalent of an *mmap*.

Implementation plans call for the object used to represent the shared memory segment to be supported by an anonymous memory-based file system. `/tmp` could be implemented as a file system of this type, potentially eliminating all I/O operations for temporary files and simply supporting them out of the processor's memory resources.

5.3. Other System Calls and Facilities

The new VM system has had an impact on other areas of the system as well, either extending or slightly altering the semantics of existing operations.

5.3.1. ''Segments''

Traditionally, the address space of a UNIX process has consisted of three segments: one each for write-protected program code (text), a heap of dynamically allocated storage (data), and the process's stack. Under the new system, a process's address space is simply a vector of pages and there exists no real structure to the address space. However, for compatibility purposes, the system maintains address ranges that ''should'' belong to such segments to support operations such as extending or contracting the data segment's ''break''. These are initialized when a program is initiated with *exec*.

5.3.2. *exec*

exec overlays a process's address space with a new program to be executed. Under the new system, *exec* performs this operation by performing the internal equivalent of an *mmap* to the file containing the program. The text and initialized data segments are mapped to the file, and the program's uninitialized data and stack areas are mapped to unnamed objects in the system's virtual memory. The boundaries of the mappings it establishes are recorded as representing the traditional ''segments'' of a UNIX process's address space.

exec establishes MAP_PRIVATE mappings, which has implications for the operation of *fork* and *ptrace*, as discussed below. The text segment is mapped with only PROT_READ and PROT_EXECUTE protections, so that write references to the text produce segmentation violations. The data segment is mapped as writable; however any page of initialized data that does not get written may be shared among all the processes running the program.

5.3.3. *fork*

Previously, a process created by *fork* had an address space made from a copy of its parent's address space. Under the new system, the address space is not copied, but the mappings defining it are. Since *exec* specifies MAP_PRIVATE on all the mappings it performs, parent and child thus effectively have copy-on-write access to a single set of objects. Further, since the mapping is generally far smaller than the data it describes, *fork* should be considerably more efficient. Any MAP_SHARED mappings in the parent are also MAP_SHARED in the child, providing the opportunity for both parent and child to operate on a common object.

5.3.4. *vfork*

Berkeley-based systems include a "VM-efficient" form of the *fork* system call to avoid the overhead of copying massive processes that simply threw away the copy operation with a subsequent *exec* call. At one point we hoped that the efficiencies gained through a reimplemented *fork* would obviate the need for *vfork*. Unfortunately, *vfork* is defined to suspend the parent process until the child performs either an *exec* or an *exit* and to allow the child full access to the parent's address space (*not* a copy) in the interim. A number of programs take advantage of this quirk, allowing the child to record data in the address space for later examination by the parent. Eliminating *vfork* would break these programs, a fact we discovered in numerous ways when early versions of the system simply treated a *vfork* as *fork*. Further, *vfork* remains fundamentally more efficient than even a *fork* that only copies an address space map, since *vfork* copies nothing.

However, to encourage programmers at Sun to avoid the use of *vfork*, we took our time restoring it to the system and as a result got many programs "fixed".

5.3.5. *ptrace*

In previous versions of the system, the *ptrace* system call (used for process debugging) would refuse to deposit a breakpoint in a program that was being run by more than one process. This restriction was imposed by the nature of the old system's facility for sharing program code, which was to share the entire text portion of an executable file.

In the new system, the system simply shares file pages among all those who have mappings to them. When a mapping is made MAP_PRIVATE, writes by a process to a page to which writes are permitted are diverted to a copy of the page – leaving the original object unaffected. *ptrace* takes advantage of the fact that an *exec* establishes the mapping to the file containing the program and its initialized data as MAP_PRIVATE, as it inserts a breakpoint by making a read-only page writable, depositing the breakpoint, and restoring the protection. The page on which the breakpoint is deposited, and only that page, is no longer shared with other users of the program – and their view of that page is unaffected.

5.3.6. *truncate*

The *truncate* system call has been changed so that it sets the length of a file. If the newly specified length is shorter than the file's current length, *truncate* behaves as before. However, if the new length is longer, the file's size is increased to the desired length. When writing a file exclusively through mapping, extending through *truncate* is the only alternative to MAP_RENAME operations for growing a file.

5.3.7. Resource Limits

Berkeley-based systems include functions for limiting the consumption of certain system resources. We have introduced a new resource limit: RLIMIT_PRIVATE. This limit controls the amount of "private memory" that a process may dynamically allocate from the system's source of unnamed backing store. In many respects, RLIMIT_PRIVATE really describes the limit that RLIMIT_DATA and RLIMIT_STACK attempt to capture, namely the amount of swap space a given process may consume.

6. Internal Interfaces

The new VM system provides a set of abstractions and operations to the rest of the kernel. In many cases, these are used directly as the basis for the system call interfaces described above. In other areas they support internal forms of those system call interfaces, allowing the kernel to perform mappings for the address space in which it operates. The VM system also relies on services from other areas of the kernel.

6.1. Internal Role of VM

In general, the kernel uses the VM system as the manager of a logical cache of memory pages and as the object manager for "address space objects". In its role as cache manager, the VM system also manages the physical storage resources of the processor, as it uses these resources to implement the cache it maintains. The VM system is a particularly effective cache manager, and maintains a high degree of sharing over multiple uses of a given page of an object. As such, it has subsumed the functions of older data structures, in particular the text table and disk block data buffer cache (the "buffer cache"). The VM system has replaced the old fixed-size buffer cache with a logical cache that uses all of the system's pageable physical memory. Thus its use as a "buffer cache" in the old sense dynamically adapts to the pattern of the system's use – in particular if the system is performing a high percentage of file references, all of the system's pageable physical memory is devoted to a function that previously only had approximately 10% of the same resources. The VM system is also responsible for the management of the system's memory management hardware, although these operations are invisible to the machine-independent portions of the kernel.

Kernel algorithms that operate on logical quantities of memory, such as the contents of file pages, do so by establishing mappings from the kernel's address space to the object they wish to access. Those algorithms that implement the *read* and *write* system calls on such memory objects are particularly interesting: they operate by creating a mapping to the object and then copying the data to or from user buffers as appropriate. When mapping is used in this manner, users of the object are provided with a consistent view of the object, even if they mix references through mapped accesses or the *read* and *write* system calls. Note that the decision to use mapping operations in this way is left to the manager of the object being accessed.

The VM system does not know the semantics of the UNIX operating system. Instead, those properties of an address space that are the province of UNIX, such as the notions of "segments" and stack-growth, are implemented by a layer of UNIX semantics over the basic VM system. By providing only the basic abstractions from the VM system itself, we believe we have made it easier to provide future system interfaces that may not have UNIX-like characteristics.

The VM system relies on the rest of the system to provide managers for the objects to which it establishes mappings. These managers are expected to provide advice and assistance to the VM system to ensure efficient system management, and to perform physical I/O operations on the objects they manage. These responsibilities are detailed further below.

6.2. *as* layer

The primary object managed by the VM system is a (process) *address space* (*as*). The interfaces through which the system requests operations on an *as* object are summarized in Table 2, and are collectively referred to as the *as*-layer of the system. An *as* contains the memory of the mappings that comprise an address space. In addition, it contains a *hardware address translation* (*hat*) structure that

Table 2 – *as* operations

Operation	Function
`struct as *as_alloc()`	*as* allocation.

`struct as *as_dup(as)`
`  struct as *as;`
	Duplicates `as` – used in *fork*.

`void as_free(as)`
`  struct as *as;`
	as deallocation.

`enum as_res`
`as_map(as, addr, size, crfp, crargsp)`
`  struct as *as; addr_t addr; u_int size;`
`  int (*crfp)(); caddr_t crargsp;`
	Internal *mmap*. Establish a mapping to an object using the mapping manager routine identified in `crfp`, providing object specific arguments in the opaque structure `crargsp`.

`enum as_res`
`as_unmap(as, addr, size)`
`  struct as *as; addr_t addr; u_int size;`
	Remove a mapping in `as`.

`enum as_res`
`as_setprot(as, addr, size, prot)`
`  struct as *as; addr_t addr;`
`  u_int size, prot;`
	Alter protection of mappings in `as`.

`enum as_res`
`as_checkprot(as, addr, size, prot)`
`  struct as *as; addr_t addr;`
`  u_int size, prot;`
	Determine whether mappings satisfy protection required by `prot`.

`enum as_res`
`as_fault(as, addr, size, type, rw)`
`  struct as *as; addr_t addr; u_int size;`
`  enum fault_type type; enum seg_rw rw;`
	Resolves a fault.

`enum as_res`
`as_faulta(as, addr, size)`
`  struct as *as; addr_t addr; u_int size;`
	Asynchronous fault – used for ''fault-ahead''.

holds the state of the memory management hardware associated with this address space. This structure is opaque to much of the VM system, and is interpreted only by a machine-dependent layer of the system, described further below.

An *as* exists independent of any of its uses, and may be shared by multiple processes, thus setting the stage for future integration of a multi-threaded address space capability as described in [KEPE 85]. The ''address space'' in which the kernel operates is also described by an *as* structure, and is the handle by which the kernel effects internal mapping operations using *as_map*.

The operations permitted on an *as* generally correspond to the functions provided by the system call interface. An implication of this is that just about any operation that the kernel could perform on an address space could also be implemented by an application directly. More work is necessary to define an interface for obtaining information about an *as*, to support the generation of *core* files, and the as-yet unspecified interfaces for reading mappings. An additional interface is also needed to support any advice operations we might choose to define in the future.

Internally to an address space, each individual mapping is treated as an object with a ''mapping object manager''. Such mappings are run-length compact encodings describing the mapping being performed, and may or may not have per-page information recorded depending on the nature of the mapping or subsequent references to the object being mapped. Due to a regrettable lack of imagination at a critical junction in our design, these ''mapping objects'' are termed *segments*, and their managers are called "segment drivers".

6.3. *hat* layer

As previously noted, a *hat* is an object representing an allocation of memory management hardware resources. The set of operations on a *hat* are not visible outside of the VM system, but represent a machine-dependent/independent boundary called the *hat*-layer. Although it provides no services to the rest of the system, the *hat*-layer is of import to those faced with porting the system to various hardware architectures. It provides the mapping from the software data structures of an *as* and its internals to those required by the hardware of the system on which it resides.

We believe that the *hat*-layer has successfully isolated the hardware-specific requirements of Sun's systems from the machine-independent portions of the VM system and the rest of the kernel. In particular, under the old system the addition of support for a virtual address cache permeated many areas of the system. Under the new system, support for the virtual address cache is isolated within the *hat* layer.

6.4. I/O Layer

The primary services the VM system requires of the rest of the kernel are physical I/O operations on the objects it maps. These operations occur across an interface called the "I/O Layer". Although used mainly to cause physical page frames to be filled (page-in) or drained (page-out) operations, the I/O layer also provides an opportunity for the managers of particular objects to map the system-specific page abstraction used by the VM system to the representation used by the object being mapped.

For instance, although the system operates on page-sized allocations, the 4.2BSD UNIX file system [MCKU 84] operates on collections of disk blocks that are often not page-sized. Efficient file system performance may also require non-page-sized I/O operations, in order to amortize the overhead of starting operations and to maximize the throughput of the particular subsystem involved. Thus, the VM system will pass several operations (such as the resolution of a fault on an object address, even one for which the VM system has a cached copy) through the object manager to provide it the opportunity to intercede. The object manager for NFS files uses these intercessions to prevent cached pages from becoming stale. Managers for network-coherent objects enforce coherence through this technique.

The I/O layer is to some extent bi-directional, as a given operation requested by the VM system may cause the object manager to request several VM-based operations. I/O clustering is an example of this, where a request by the VM system to obtain a page's worth of data may cause the object manager to actually schedule an I/O operation for logical pages surrounding the one requested in the hopes of avoiding future I/O requests. The old notion of "read-ahead" is implemented in this manner, and each object manager has the opportunity to recognize and act on patterns of access to a given object in a manner that maximizes its performance.

7. Project Status & Future Work

The architecture described in this paper has been implemented and ported to the Sun-2 and Sun-3 families of workstations. At present, all our major functional goals have been met. The work has consumed approximately four man-years of effort over a year and a half of real time. A surprisingly large amount of effort has been drained by efforts to interpose the VM system as the logical cache manager for the file systems, in particular with respect to the 4.2BSD UNIX file system.

With respect to our performance goals, more tuning work is required before we can claim to meet them. However, in some areas dealing with file access, early benchmarks reveal substantial performance improvements resulting from the much larger cache available for I/O operations. We expect further performance improvements when more of the system uses the new mechanisms. In particular, we expect an implementation of shared libraries to have a substantial impact upon the use of system resources. Future uses of mapping include a rewritten standard I/O library to use *mmap* rather than *read* and perhaps *write*, thus eliminating the dual copying of data and providing a transparent performance improvement to many applications. As sharing increases in the system, we expect the requirements for swap resources to decrease.

Other future work involves refining and completing the interfaces that have not yet been fully defined. We plan an investigation of new management policies, especially with respect to different page-replacement policies and the better integration of memory and processor scheduling. We would also like to port the system to different hardware bases, in particular to the VAX, to test the success of the *hat* layer in isolating machine dependencies from the rest of the system.

8. Conclusions

We believe the new VM architecture successfully meets our goals. Reviewing these reveals:

- **Unify memory handling.** All VM operations have been unified around the single notion of file mapping. Extant operations such as *fork* and *exec* have been reconstructed and their performance, and in some cases function, has been improved through their use of the new mechanisms.
- **Non-kernel implementation of many functions.** Although we were disappointed that kernel support was required to implement System V shared memory segments, we believe that this goal has been largely satisfied. In particular, our implementation of shared libraries [GING 87] requires no specific kernel support. We believe the basic operations the interfaces provide will permit the construction of other useful abstractions with user-level programming.
- **Improved portability.** Although more experience is required, we were pleased with the degree to which the Sun-3 virtual address cache was easily incorporated into the new system, in comparison with the difficulty experienced in integrating it into the previous system.
- **Consistent with environment.** The new system builds on the abstractions already in UNIX, in particular with respect to our use of the UNIX file system as the name space for named virtual memory objects. The integrated use of the new facilities in the system has helped to extend the previous abstractions in a natural manner. The semantics offered by the basic system mechanisms also do not impede the heterogeneous use of objects accessed through the system, an important consideration for the networked environments in which we expect the system to operate.

Finally, we have provided the functionality that motivated the work in the first place.

9. Acknowledgements

The system was designed by the authors, with Joe Moran providing the bulk of the implementation. Bill Joy offered commentary and advice on the architecture, as well as insights into the intents of the 4.2BSD interface, and an initial sketch of an implementation of the internal VM interfaces. Kirk McKusick and Mike Karels of UC Berkeley, CSRG, spent several days discussing the issues with us. The other members of Sun's System Software group gave considerable assistance and advice during the design and implementation of the system.

10. References

[ACCE 86] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[AT&T 86] AT&T, *System V Interface Definition*, Volume I, 1986

[BOBR 72] Bobrow, D. G., J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10", *Communications of the ACM*, Volume 15, No. 3, March 1972.

[GING 87] Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, "Shared Libraries in SunOS", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[JOY 83] Joy, W. N., R. S. Fabry, S. J. Leffler, M. K. McKusick, *4.2BSD System Manual*, Computer Systems Research Group, Computer Science Division, University of

California, Berkeley, 1983.

[KEPE 85] Kepecs, J. H., ''Lightweight Processes for UNIX Implementation and Applications'', *Summer Conference Proceedings, Portland 1985*, USENIX Association, 1985.

[KLEI 86] Kleiman, S. R., ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX'', *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[MKCU 84] McKusick, M. K., W. N. Joy, S. J. Leffler, R. S. Fabry, ''A Fast File System for UNIX'', *Transactions on Computer Systems*, Volume 2, No. 3, August 1984.

[MURP 72] Murphy, D. L., ''Storage organization and management in TENEX'', *Proceedings of the Fall Joint Computer Conference*, AFIPS, 1972.

[ORGA 72] Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, 1972.

Virtual Address Cache in UNIX†

Ray Cheng

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, Ca. 94043
rcheng@Sun.COM or sun!rcheng

Most of the cache memories in computer systems are addressed by physical addresses. In systems that support virtual addresses, this means that the cache is accessible after a virtual to physical translation is done. In order to reduce the cache access time, Sun-3 Series 200 workstations include a cache accessed by virtual addresses. However, unlike physical address caches, virtual address caches are not transparent to software. Data consistency problems arise when there is a change to the virtual to physical mapping or when two or more virtual addresses map to the same physical address.

To hide this data consistency problem from application programs, the kernel should ensure that a program runs on a machine with a virtual address cache produces the same result as the one it produces when it runs on a machine without a virtual address cache. To guarantee such system correctness, the kernel maintains the following invariant: if an entry is in the cache, its virtual to physical mapping must be correct and no other virtual address maps to that physical address without going through the same cache entry.

There are three things the kernel can do to satisfy this condition. The first is to flush an entry from the cache when its virtual to physical mapping becomes incorrect, e.g. when the mapping of u page of the out-going process becomes incorrect during a context switch. The second is to make virtual addresses differ by modulo 128K when we set up a virtual address that maps to the same physical address as another virtual address, e.g. in the implementation of AT&T System V shared memory. Finally, if the kernel doesn't know when to flush the cache and the assignment of virtual addresses that map to the same physical address is beyond the kernel's control, e.g. in the implementation of the mmap routines in 4.2BSD UNIX, the kernel makes these virtual addresses non-cacheable.

This paper first gives some background in cache memories and the Sun-3 cache architecture in particular. It then describes the approach we took and the new routines we added to the kernel. Several examples illustrate why and how some entries are flushed from the cache when a mapping is invalidated, when a mapping becomes invalid implicitly, and when the protection attributes of a mapping are changed. Since flushing an entry from the cache takes up to several hundred microseconds, we avoid cache flushing as much as possible. This paper describes several cases where cache flushings are avoided even when the mappings become incorrect. After that, the uses of modulo 128K addressing and non-cache pages are discussed. The debugging turned out to be as difficult as we had feared, especially due to the fact that we debugged the kernel and the cache hardware at the same time. Finally, the overhead of cache flushings as well as their measurement are discussed.

† UNIX is a registered trademark of AT&T

1. Introduction

To increase effective memory access speed, modern computer systems use a cache memory in front of main memory. Almost all computer systems address cache memories by physical addresses. This means that a Translation Lookaside Buffer (TLB) lookup and possibly a virtual-to-physical translation is needed before the information in the cache memory can be accessed.

In Sun-3 200 series machines, the cache memory is addressed by virtual addresses. The advantage is that TLB is eliminated and the cache can be accessed without a virtual-to-physical mapping. However, this cache is not transparent to software. To relieve users of the need to modify their programs, the operating system has to make the cache transparent to user level programs. In this paper we present our experience with the extension of a 4.2BSD based UNIX kernel for a machine with such a cache memory.

2. Background

Cache memories are small, high speed memories used to hold information that is believed to be currently in use [Smith, 82]. Information located in cache memories can be accessed in much less time than that located in the main memory. Thus, a CPU with a cache memory spends less time waiting for instructions or data to be fetched or stored.

When a write operation is performed, main memory can be updated in two fashions. In the first method, the cache memory receives the write and the main memory is not updated until that cache line is replaced. This method is known as copy-back (or write-back). The second method, known as write-through, updates both the cache memory and the main memory when the write is performed.

Cache memories can be addressed either by physical addresses or by virtual addresses. The cache memory that is addressed by physical addresses is called a physical address cache while the cache memory that is addressed by virtual addresses is called a virtual address cache. Their conceptual difference relative to the virtual-to-physical address translation is illustrated in Figure 1.

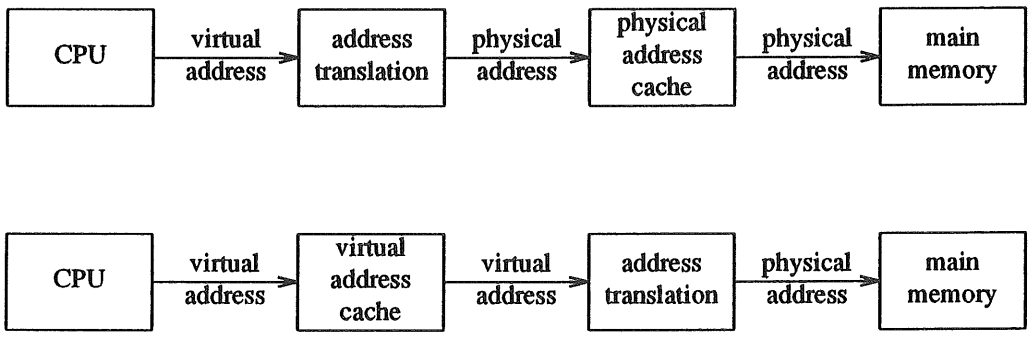


Figure 1 physical address cache vs. virtual address cache

3. Sun-3 Cache Architecture

The Sun-3 virtual memory architecture provides each process with a 256 megabytes virtual address space [Sun, 86]. The Sun-3 MMU uses a page size of 8K bytes and segment size of 128K bytes. The MMU consists of eight distinct address spaces or "contexts", each of them has a size of 256M bytes. The kernel is responsible for multiplexing the hardware resource of contexts among the set of processes it supports, using each context to represent a process's virtual address space.

The Sun-3 cache is a virtual address, write-back cache. The cache is 64K bytes in size with 16 byte lines. It has a context field in its cache tags to distinguish the eight contexts of the Sun-3 MMU. Hence the entire cache is not wiped out on a context switch. When a virtual-to-physical mapping is set up in the MMU, a page may be made non-cacheable. If a page is made non-cacheable in the MMU, the information in this page will not be put in the cache.

To make it possible for the system to work correctly with the virtual address cache, the architecture includes three cache flush operations: the page match flush, the segment match flush, and the context match flush. These flush operations flush all cache lines whose tags match a page address, a segment address, or a context number. When a cache line is flushed, if that line is modified and valid, a write-back to the main memory is done. In addition, if the line is valid, it becomes invalid after being flushed. (A hit occurs when the requested virtual address matches with the virtual address of a *valid* line as well as the type of the access satisfies the protection check.)

Further, the Sun-3 cache guarantees that all virtual addresses that map to the same physical address are put to a common cache location if their (virtual) addresses differ by a multiple of 128K bytes. This applies to virtual addresses within the same context or between different contexts.

4. Problems with Virtual Address Caches

The correctness criterion for introducing a cache is that any program run on a system with a cache should produce the same result as it produces from a system without such a cache. A system with a virtual address cache introduces two kinds of data consistency problems: *mapping change* and *synonyms.*

4.1. Mapping Changes

Mapping change introduces data inconsistency as follows (Figure 2): At time t1, virtual address v maps to physical address p1 and there is a write operation that writes value x to the content of virtual address v. This value x is associated with virtual address v in the cache. At a later time t2, the mapping of v is changed such that it maps to physical address p2 which contains a value y. Then, the CPU issues a read operation to v. On systems without such a cache, the value y should be the result of this read access. However, if <v, x > is valid in the cache, the value x will be the result of the read access. Furthermore, the value x is not written to the physical memory p1.

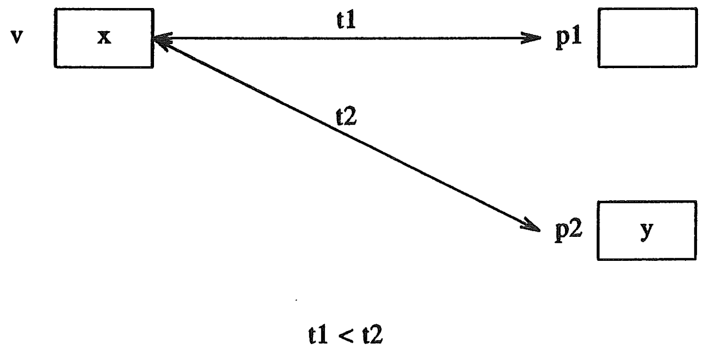


Figure 2 Data Inconsistency due to Mapping Changes

4.2. Synonyms

Synonym is the case when there is more than one virtual address mapped to the same physical address. Synonyms introduce another kind of data inconsistency as follows: Virtual addresses v1 and v2 both map to physical address p. At time t1, the CPU writes the value *x* to v1. At time t2 > t1, the CPU writes the value *y* to v2. (Figure 3) At this time the value *x* is associated with v1 in the cache while the value *y* is associated with v2 in the cache. The physical memory p should contain the value *y* since it is written at the later time. Next, at time t3 > t2 the CPU reads from v1. Since <v1, *x* > is stored in the cache, the value *x* is returned to the CPU. However, on systems without a virtual address cache, the value *y* is returned to the CPU.

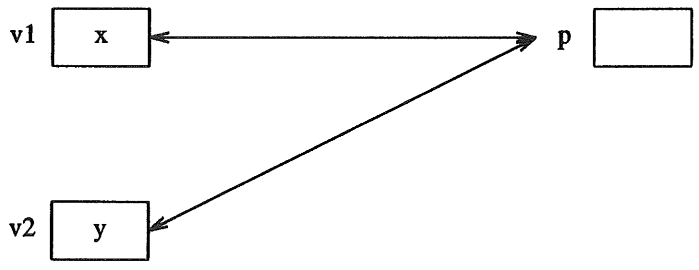


Figure 3 Data Inconsistency due to Synonyms

5. Unix Kernel Extensions

The goal of the kernel extensions is to ensure that any user level program running on machines with a virtual address cache produces the same result as it would produce when running on machines without such a cache. Also, attention is paid to introduce as little cache-induced overhead as possible. Such overhead can be the result of cache flush operations or of non-cacheable page accesses.

The kernel uses cache flush operation to solve the data inconsistency problem introduced by mapping changes. For synonyms, the kernel tries to utilize the modulo 128K feature of the Sun-3 cache architecture, to execute cache flush operations, and to make pages non-cacheable. Since cache flush operations take tens to hundreds of microseconds, we attempted to avoid such operations as much as we could.

5.1. Mapping Changes

This data inconsistency problem can be solved if the kernel flushes the affected portion of the cache whenever there is a mapping change. As shown in Figure 4, before the mapping v-to-p1 is changed to v-to-p2, we issue a cache flush operation to virtual address v. Then, the value *x* is written physical memory p1 and there is no valid entry for virtual address v in the cache. When the CPU reads from virtual address v after the mapping v-to-p2 is set up, since v doesn't have a valid entry in the cache, the read causes a cache miss and the value *y* is obtained from the physical memory.

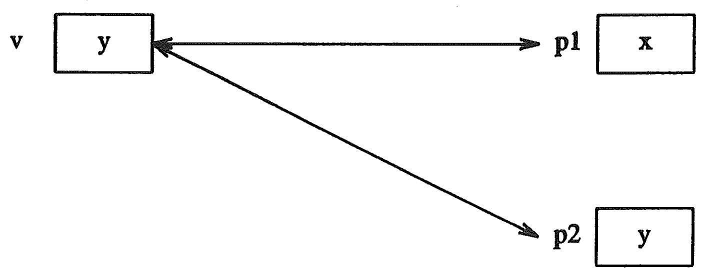


Figure 4 Mapping Changes with cache flushes

If the mapping of v-to-p1 is invalidated before the mapping v-to-p2 is set up, as occurs in lots of places in 4.2BSD Unix, we only have to flush the cache when a mapping becomes invalid. The mapping change from invalid to v-to-p2 doesn't need a cache flush, because there is no valid entry for virtual address v in the cache. We generalized this cache flushing strategy as follows: There is always a cache flush operation when a mapping is changed from valid to invalid, but there is no cache flush operation when a mapping is changed from invalid to valid. This saves a number of unnecessary cache flush operations while maintaining the correctness criteria.

Example 1. In SunOS Release 3.2, the u page is in the kernel virtual address space. This kernel virtual address maps to the physical u page of the running process. Therefore, when a process is schedule to run during a context switch, a mapping from virtual address _u to the physical address of its u page has to be set up. Similarly, such a mapping is invalidated when a process is "switched" out during context switch. To avoid the data inconsistency due to this mapping change during context switch, we do a page match flush when the mapping of u page is invalidated. (We don't do a page match flush when a new mapping for the u page is being set up.)

Example 2. In pageout(), the pageout daemon marks not-recently-used pages to be invalid. If there is no page match flush for this page and the MMU mapping is invalidated accordingly, a subsequent write-back of this page to the physical memory will fail.

There are also cases where a mapping is not released or invalidated explicitly. To follow our cache flushing discipline, namely, flush when a mapping becomes invalid not when a mapping becomes valid, we deem the mapping to be invalid when this mapping is not used any more and start the flush operation at this time. For example, forkutl is used to map to the physical u page of the child process when the kernel is servicing the fork(2) system call. This mapping is not released explicitly when the routine using forkutl returns. However, we deem the mapping to be invalid before the routine returns. Therefore, the page match flush for forkutl is done before the routine returns.

5.2. Synonyms

If the virtual addresses in a case of synonym can be set by the kernel, such virtual addresses are set such that their differences are modulo 128K. Thus, the Sun-3 cache architecture guarantees that all such virtual addresses that map to the same physical address occupy the same cache line in the cache memory. This solves the data inconsistency problem without any cache flush overhead. This technique was used in the implementation of the System V shared memory in SunOS Release 3.2.

Many times it is inconvenient to assign values to virtual addresses. For example, it is impractical to allocate kernel global variables that may be used to map to same physical addresses at addresses that differ by a multiple of 128K. However, if the kernel knows about the access pattern of synonyms v1 and v2, we can treat the other mapping as invalid while one mapping is actively in use. Then we can flush the virtual address when its mapping becomes invalid. The following example illustrates this method.

Example 3. In SunOS Release 3.2, a Direct Virtual Memory Access (DVMA ™) operation from virtual address v1 starts with the kernel setting up a mapping from another virtual address v2 in the DVMA region to the same physical address as v1 maps to. Then, the requested DVMA operation, either read or write, is started through virtual address v2. Thus, both v1 and v2 map to the same physical address, a case of synonyms.

However, in this case, we know that v1 is accessed first, then v2 is accessed, and finally v1 is accessed again (Figure 5). When the mapping of v2-to-p is being set up, we view the mapping of v1-to-p as invalid and do a page match flush on v1. When v2 is not accessed any more for this DVMA operation, we view that the mapping of v2-to-p becomes invalid and do a page flush on v2. All these flush operations turned out to be necessary both on DVMA read operations and on DVMA write operations.

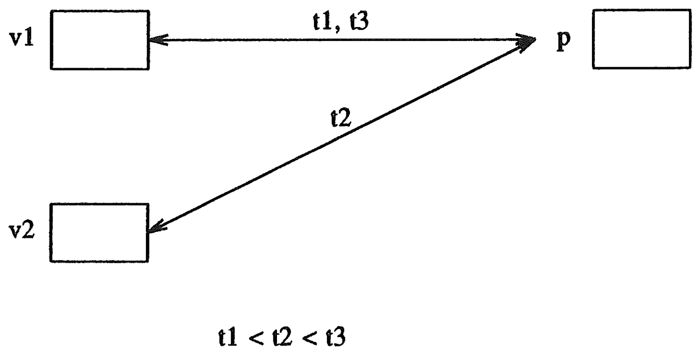


Figure 5 Synonyms with cache flushes

5.3. Don't Cache Page

If in a case of synonyms, we can neither set the virtual addresses nor know about the access pattern of the virtual addresses, we make all these virtual pages non-cacheable. As discussed in Section 3, any page that is made non-cacheable in the MMU is not included in the cache. On Sun-3 200 series implementation, when a page is non-cacheable, its access is much slower than the the speed CPU can access memory. Thus, a number of wait states are needed. Therefore, this method is used as the last resort to guarantee system correctness.

Example 4. The mmap(2) routine from 4.2BSD Unix allows different user level programs to map to the same physical address. In this case, user level addresses are determined by user programs and their access behavior is unknown to the kernel. As a result, the kernel makes these user pages non-cacheable. If the kernel page is also used by a device driver, the device driver should make the shared kernel page non-cacheable also (Figure 6).

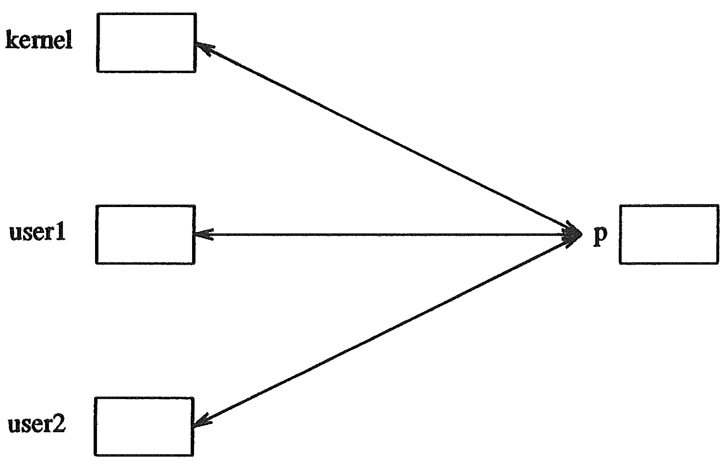


Figure 6 Synonyms that Require the Use of Don't Cache Pages

5.4. Don't Flush if Not Necessary

The cache flush operations are rather time consuming in our implementation. Therefore, we avoid flushing as much as possible. One trick in the case of u page flush is that since only half of an 8K bytes pages is really used, the kernel flushes 4K bytes instead of flushing the entire page. There are a number of cases where flush operations can be avoided when a mapping changes from valid to invalid. For example in swap() of 4.2BSD Unix, dirty pages are mapped to the context of proc[2] and hence invalidate the previous mapping. Since dirty pages have been flushed in pageout() already, there is no need to flush these pages again in swap().

6. Performance

In order to measure the cache-induced overhead, we added instrumentation code to the kernel to record the total number of each kind of flush. Then we wrote a user level program that reads these kernel numbers. Next, before and after we ran benchmark programs we probed the number of each type of flushes. The differences of these numbers approximate the number of each kind of flush occurred from running this benchmark program.

Though hardware engineers are able to give us the minimum time needed to do each kind of flushes, the time that a flush really takes depends on the number of lines being modified at the time of flush and on other system activities such as Ethernet traffic and VMEbus TM activities. Lacking analytical data, we decided to use a logical analyzer to measured the average time needed to do a flush. Since different benchmarks cause the cache lines modified quite differently, we measure the average flush time separately for each benchmark we ran. We also estimated the time spent in software to instruct the hardware to do a flush. Next we multiplied the number of each kind of flushes by the average time of each kind of flushes to get an approximate total time spent in flushing the cache for a benchmark program. Lastly, we divided the time spent in flushing the cache by the total amount of time spent in running the benchmark program.

We ran one benchmark program at a time on lightly loaded multi-user mode. For the dhrystone benchmark, only 0.13% of total time was spent in flushing the cache. Most of the flushes were to flush the u page during context switches. Also, the u page was barely modified hence the cache hardware spent minimal time doing write-back. Another benchmark program which causes page faults to occur continuously spent 3.0% of total time in flushing the cache. In this benchmark, paging to and from the disk using DVMA operations caused many page match flushes. Also, almost all of such flushes needed write-back operations.

7. Conclusions

To guarantee the correctness of the system, we keep the following invariant condition true at all time. Throughout the design, this invariant condition was checked against to decide whether a cache flushing was needed.

If an entry is in the cache, its mapping (including protection violation check) from the virtual address to physical address must be correct.

Mappings are changed, both implicitly and explicitly, in many different places in the 4.2BSD kernel. It would be easier to identify all mapping changes if they were placed only in a few routines.

The debugging was as tricky as we had feared, worsened by the fact that we debugged the kernel and the cache hardware with the kernel at the same time. When the system crashed, the cause could be that the kernel missed a cache flush a short time ago in this context, or that the kernel missed a cache flush several context switches back. It also could be that somewhere the kernel flushed at the right time but flushed a wrong address. Still, as it sometimes turned to be, it could also be that the cache hardware didn't flush the cache as it should.

Finally, it was nice to see the system ran faster than we had anticipated. Also, the flush overhead was found to be smaller than we expected.

Acknowledgements

Ed Hunter provided enormous help in the debugging stage, especially in using logical analyzer to identify hardware bugs to the hardware people. Jo Moran and Bill Shannon helped to find the last (known) bug.

8. References

[Smith, 82]
Smith, Alan Jay, "Cache Memories", *ACM Computing Surveys*, September 1982.

[Sun, 1986]
Sun Microsystems, Inc., "Sun-3 Architecture: A Sun Technical Report" August 1986

;login:

The USENIX Association Newsletter

Volume 12, Number 5 **September/October 1987**

CONTENTS

The closing date for submissions for the next issue of *;login:* is October 30, 1987

THE PROFESSIONAL AND TECHNICAL UNIX® ASSOCIATION

Computer Graphics Workshop

Boston Marriott Cambridge
Cambridge, MA

October 8-9, 1987

The Fourth USENIX Computer Graphics Workshop will be held at the Boston Marriott Cambridge in Cambridge, MA, October 8 and 9, 1987, with a no-host reception on the evening of October 7.

Registration will be $200 per attendee and must be paid in advance. There will be no on-site registration.

There is a special hotel rate for workshop attendees of $115 per night, single or double. Call the Marriott direct for reservations: 617-494-6600. Be sure to mention that you are a USENIX Workshop attendee. The Marriott has a strict cut-off of September 16 for its special rate. Reservations made after that date will be on a space and rate available basis.

For further program information, contact:

Tom Duff at *research!td* or Lou Katz at *ucbvax!lou.*

POSIX Portability Workshop

Berkeley Marina Marriott
Berkeley, CA

October 22-23, 1987

This USENIX workshop will bring together system and application implementors faced with the problems, "challenges," and other considerations that arise from attempting to make their products compliant with IEEE Standard 1003.

The first day of the workshop will consist of presentations of brief position papers describing experiences, dilemmas, and solutions. On the second day it is planned to form smaller focus groups to brainstorm additional solutions, dig deeper into specific areas, and attempt to forge common approaches to some of the dilemmas.

For further program information, contact:

Jim McGinness
(603) 884-5703
decvax!jmcg or jmcg@decvax.DEC.COM

C++ Workshop

Eldorado Hotel
Santa Fe, NM

November 9-10, 1987

USENIX is pleased to be hosting a workshop on the C++ language. This two-day event will bring together users of the C++ language to share their experiences. The program will be somewhat informal, comprising a mix of full presentations and shorter talks.

Bjarne Stroustrup, designer of the C++ language, has agreed to speak on topics guaranteed not to be in the C++ book, including multiple inheritance and other futures.

In conjunction with the workshop, USENIX will publish two documents:

Presentation Summaries: one or two pages from each speaker. A free copy will be available on-site to each registrant.

Full Proceedings: The proceedings will be mailed free to each registrant and will be available to the public through the USENIX office.

Vendors of products related to C++ are encouraged to send technical personnel to demonstrate their products. Call the USENIX Conference Coordinator below for more information.

The Conference Chair:

Keith Gorlen
Building 12A, Room 2017
National Institutes of Health
Bethesda, MD 20892

301-496-5363
usenix!nih-csl!keith

For registration or hotel information for any USENIX workshop, contact:

Judith F. DesHarnais
USENIX Conference Coordinator
P.O. Box 385
Sunset Beach, CA 90742

(213) 592-3243
usenix!judy

Call for Papers
Winter 1988 USENIX Conference

Dallas, Texas

February 9-12, 1988

Please consider submitting an abstract for your paper to be presented at the Winter 1988 USENIX conference. Abstracts should be around 250-750 words long and should emphasize what is new and interesting about the work. The final typeset paper should be 8-12 pages long.

The Winter conference will be four days long: two days of tutorials only and two days of papers only.

Suggested topic areas for this conference include (but are not limited to):

- Electronic Publishing
- Novel Kernels
- New Software Tools
- New Applications
- System Administration
  (including distributed systems and integrated environments)
- Security in UNIX
- Future Trends in UNIX

This conference may include a "miscellaneous" session which will include those papers which do not fit into standard tracks. Vendor presentations should contain technical information and be of interest to the general community.

Abstracts are due by **October 23, 1987**; papers absolutely must be submitted by **January 4, 1988**. Notifications of acceptance of abstracts will be sent out by November 6. Papers that do not meet the promise of their abstract will be rejected. Talks will be given on all papers published in the *Proceedings*; failure to submit a paper for an abstract will result in forfeiture of the talk.

Please contact the program chairman for additional information:

Rob Kolstad	214-952-0351 (W)
CONVEX Computer Corporation	214-690-1297 (H)
701 Plano Road	214-952-0560 (FAX)
Richardson, TX 75081	

{usenix,ihnp4,uiucdcs,allegra,sun}!convex!kolstad

Please include your network address (if available) with all correspondence. It should be an ARPANET (EDUNET, COMNET), BITNET, or CSNET address or a UUCP address relative to a well-known host (e.g., *mcvax, ucbvax, decvax*, or, *ihnp4*).

Call for Papers
Summer 1988 USENIX Conference

San Francisco

June 20-24, 1988

Papers in all areas of UNIX-related research and development are solicited for formal review for the technical program of the 1988 Summer USENIX Conference. Accepted papers will be presented during the three days of technical sessions at the conference and published in the conference proceedings. The technical program is considered the leading forum for the presentation of new developments in work related to or based on the UNIX operating system.

Appropriate topics for technical presentations include, but are not limited to:

- Kernel enhancements
- UNIX on new hardware
- User interfaces
- UNIX system management
- The internationalization of UNIX
- Performance analysis and tuning
- Standardization efforts
- UNIX in new application environments
- Security
- Software management

All submissions should contain new and interesting work. Unlike previous technical programs for USENIX conferences, the San Francisco conference is requiring the submission of **full papers** rather than extended abstracts. Further, a tight review and production cycle will not allow time for rewrite and re-review. (Time is, however, scheduled for authors of accepted papers to perform minor revisions.) Acceptance or rejection of a paper will be based *solely* on the work as submitted.

To be considered for the conference, a paper should include an abstract of 100 to 300 words, a discussion of how the reported results relate to other work, illustrative figures, and citations to relevant literature. The paper should present sufficient detail of the work plus appropriate background or references to enable the reviewers to perform a fair comparison with other work submitted for the conference. Full papers should be 8-12 single spaced typeset pages, which corresponds to roughly 20 double spaced, unformatted, typed pages. Format requirements will be described separately from this call. All final papers must be submitted in a format suitable for camera-ready copy. For authors who do not have access to a suitable output device, facilities will be provided.

Four copies of each submitted paper should be received by **February 19, 1988**; this is an absolute deadline. Papers not received by this date will not be reviewed. Papers which clearly do not meet USENIX's standards for applicability, originality, completeness, or page length may be rejected without review. Acceptance notification will be by **April 4, 1988**, and final camera-ready papers will be due by **April 25, 1988**.

Send technical program submissions to:

Sam Leffler	415-499-3600
SF-USENIX Technical Program	ucbvax!sfusenix
PIXAR	
P.O. Box 13719	
San Rafael, CA 94913-3719	

Second Distribution of Berkeley PDP-11[†] Software for UNIX[‡]

Release 2.10
(Revised April 1987)

The USENIX Association and the Computer Systems Research Group (CSRG) of the University of California, Berkeley, are pleased to announce the distribution of a new release of the "Second Berkeley Software Distribution" (2.10BSD).

This release will be handled by the USENIX association, and is available to all V7, System III, System V, and 2.9BSD licensees. The Association will continue to maintain the non-profit price of $200, as was charged by the CSRG. The release will consist of two 2400 foot, 1600 bpi tapes (approximately 80Mb) and approximately 100 pages of documentation. If you require 800 bpi tapes, please contact USENIX for more information.

If you have questions about the distribution of the release, please contact USENIX at:

2.10BSD
USENIX Association
P.O. Box 2299
Berkeley, CA 94710

+1 415 528-8649

USENIX may also be contacted by electronic mail at:

{ucbvax,decvax}!usenix!office

If you have technical questions about the release, please contact Keith Bostic at:

{ucbvax,seismo}!keith
keith@okeeffe.berkeley.edu

+1 415 642-4948

Q: What machines will 2.10BSD run on?

2.10BSD will run on:

11/24/34/44/53/60/70/73/83/84
11/23/35/40/45/50/55 with 18 or 22 bit addressing

2.10 WILL NOT run on:

T11, 11/03/04/05/10/15/20/21
11/23/35/40/45/50/55 with 16 bit addressing

Q: What's new in this release?

Lots of stuff. This release is 4.3BSD. We don't expect to distribute manuals this time, we expect people to simply use the 4.3BSD ones. A list of some of the larger things that have been added:

22-bit Qbus support
4.3BSD networking, (TCP/IP, SLIP)
4.3BSD serial line drivers
4.3BSD C library

† DEC, PDP, and VAX are trademarks of Digital Equipment Corporation.
‡ UNIX is a trademark of Bell Laboratories.

most of the 4.3BSD application programs
complete set of 4.3BSD system calls
MSCP device driver for (RQDX? UDA50, KLESI)
RAM disk
inode, core, and swap caching
conversion of the entire system to a 4.3BSD structure

Q: Why get this release?

You want to get this release for one of two reasons. Either you have a number of 4.3BSD programs or machines in your environment and you'd like consistency across the environment, or you want a faster, cleaner version of 2.9BSD, with or without networking.

This release is, without question, considerably faster than any other PDP-11 system out there. There have been several major changes to the 2.10BSD kernel to speed it up.

- The kernel *namei* routine has been modified to read the entire path name in at once rather than as a single character at a time, as well as maintaining a cache of its position in the current directory.
- The *exec* routine now copies its arguments a string at a time, rather than a character at a time.
- All inodes are placed in an LRU cache, eliminating going to disk for often used inodes; kernel inodes also contain more of the disk inode information to eliminate require disk access for *stat*(2) calls.
- Both core and swap are LRU cached; the former is particularly interesting on PDP-11's with large amounts (for PDP's, anyway) of memory. Our experience with an 11/44 with 4M of memory, in a student environment, is that it never swaps, and only rarely do programs leave core.

  This change is largely responsible for My Favorite Timing: Ultrix 11, V3.0, on my 11/73, with a single RD52, takes 1.1 system seconds to run *vi*. 2.9BSD takes approximately .9 system seconds, a difference probably attributable to the fact that 2.9BSD has *vfork*. Once 2.10BSD has the *vi* image in its core cache, it executes *vi* in .2 system seconds.
- Finally, many other speedups, such as rewriting several of C library routines in assembler, replacing the kernel clist routines with the faster 4.3BSD ones, caching and hashing process id's, and splitting the process list into multiple lists have been added.

Q: How good is the networking?

The networking is 4.3BSD's. It runs, it runs correctly. It eats space like there's some kind of reward. We are considering fixing the latter by moving the networking into supervisor space.

Q: Will this release be supported?

This release is not supported, nor should it be considered an official Berkeley release. It was called 2.10BSD because 2.9BSD has clearly become overworked and System V was already taken.

The "bugs" address supplied with this release (as well as with the 4BSD releases) will work for some unknown period of time; make sure that the "Index:" line of the bug report indicates that the release is "2BSD." See the *sendbug*(8) program for more details. All fixes that we make, or that are sent to us, will be posted on USENET, in the news group *comp.bugs.2bsd*. USENIX is aware of this problem and is willing to make hard-copy bug reports available to those of you not connected to the net.

To summarize, all that I can say is that any *major* problems will be fixed, i.e. if you've got a program that's crashing the kernel, we'll be inclined to fix it. If *ls* is misformatting its output, you're probably on your own.

Q: Is this the last release?

Yes, at least by us; quite frankly, we'd rather sacrifice our chance at heaven than look at a 16-bit machine again.

Q: Who did all this wonderful, exciting, neat stuff?

Mostly Casey Leedom, of California State University, Stanislaus, and Keith Bostic, of the CSRG. From the "Changes to the Kernel in 2.10BSD" paper:

> The authors gratefully acknowledge the contributions of many other people to the work described here. Major contributors include Gregory Travis of the Institute for Social Research, and Steven Uitti of Purdue University. Jeff Johnson, also of the Institute for Social Research, was largely responsible for the port of 4.3BSD's networking to 2.10BSD. Cyrus Rahman of Duke University should hold some kind of record for being able to get the entire kernel rewritten with a single 10-line bug report. Much credit should also go to the authors of 4.2BSD and 4.3BSD from which we stole everything that wasn't nailed down and several things that were. (Just *diff* this document against *Changes to the Kernel in 4.2BSD* if you don't believe that!) We are also grateful for the invaluable guidance provided by Michael Karels, of the Computer Science Research Group, at Berkeley – although we felt that his suggestion that we "just buy a VAX," while perhaps more practical, was not entirely within the spirit of the project.

The tape that USENIX will be distributing for the first few weeks will **only** support machines with split I/D and floating point hardware. This is not because any work remains to be done, but because we just haven't had the time to build and test a system.

Sites wishing to run 2.10BSD should also be aware that the networking is only lightly tested, and that certain hardware has yet to be ported. Contact Keith Bostic at the above address for current information as to the status of the networking. As of August 6, 1987, the complete 4.3BSD networking is in place and running, albeit with minor problems. The holdup is that only the Interlan Ethernet driver has been ported, as well as some major space constraints. Note, if we decide to go to a supervisor space networking, 2.10 networking will only run on:

11/44/53/70/73/83/84
11/45/50/55 with 18 bit addressing

Keith Bostic
Casey Leedom

RT PC Distributed Services: File System

Charles H. Sauer
Don W. Johnson
Larry K. Loucks
Amal A. Shaheen-Gouda
Todd A. Smith

IBM Industry System Products
Austin, Texas 78758

Introduction

RT PC Distributed Services (RT/DS) provides distributed operating system capabilities for the AIX[1] operating system. These include distributed files with local/remote transparency, a form of "single system image" and distributed interprocess communication. The distributed file design supports "traditional" AIX and UNIX[2] file system semantics. This allows applications, including data management/data base applications, to be used in the distributed environment without modification to existing *object* code. The design incorporates IBM architectures such as SNA and some of the architectures of Sun Microsystems'[3] NFS. This paper focuses on key characteristics and decisions in the design of the Distributed Services file system.

There have been numerous research efforts and a number of products with goals and characteristics similar to the RT/DS file system. Perhaps the best known of these are LOCUS [Popek *et al* 1981, Popek and Walker 1985], NFS [Sandberg *et al* 1985, Sun 1986] and RFS [Rifkin *et al* 1986]. We studied each of these, and many other, previous designs. As we describe the RT/DS file system, we will contrast our design with some of its predecessors.

Administrative Environments

Assumptions about administrative environments are fundamental to design of distributed systems, yet administrative concerns are often omitted from primary consideration. Two primitive elements can be identified in the environments we anticipate: multi-machine clusters and independently administered machines.

Multi-Machine Clusters

All of the machines[4] in a multi-machine cluster are administered uniformly, let us assume by the same person. The machines in the cluster are likely owned by the same department, and members of the department can use the machines equivalently, i.e., the multiple machines present a "single system image." Regardless of which machine in the cluster is used, login ids and passwords are the same, the same programs/data files/directories are accessible and authorization characteristics are the same. The large boxes marked D46, D29, ... in Figure 1 are meant to suggest multi-machine clusters, with each small box representing a machine. Of course, the machines may be dispersed geographically within the limitations of the networks used – they are shown together in a room for convenience in drawing the figure.

[1] AIX is a trademark of International Business Machines Corporation.

[2] Developed and licensed by AT&T. UNIX is a registered trademark in the U.S.A. and other countries.

[3] Sun Microsystems is a trademark of Sun Microsystems, Inc.

[4] We use "machine" to indicate a generic computer – a personal computer, workstation or larger computer.

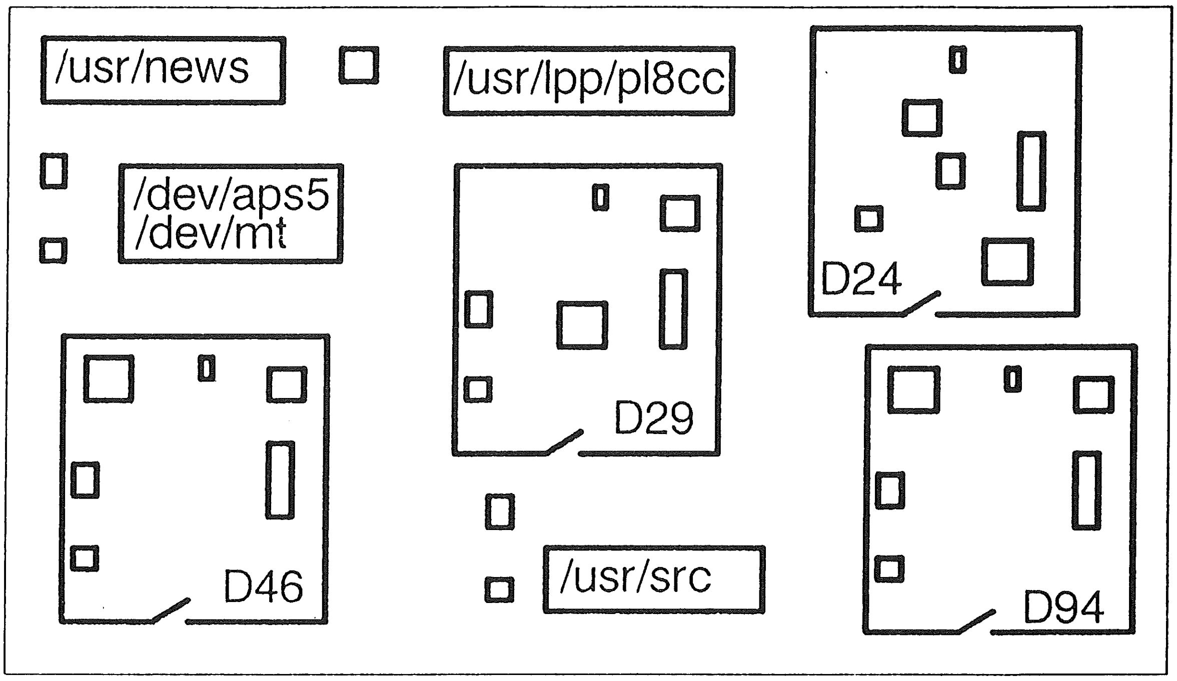


Figure 1. Multi-machine clusters and separately administered machines.

Independently Administered Machines

The other primitive element is the independently administered machine. These machines fall into two subcategories: servers and private machines. Servers in this sense are not functionally different from other servers (e.g., file or device servers) which may be found within multi-machine clusters, but they are administered independently from other machines/clusters. The boxes with path names in Figure 1 are intended to suggest file/device servers administered independently. Other machines may be independently administered because the owners are unwilling to allow others to assume administrative responsibility. Both of these subcategories can be considered degenerate cases of the multi-machine cluster, but it is convenient to discuss them separately.

Networks of Multi-Machine Clusters/Independently Administered Machines

As organizations evolve toward connecting all machines with multimegabit per second networks, administrative configurations such as the one depicted in Figure 1 will inevitably occur. It will be required that all of the machines be able to communicate with one another, and a high degree of network transparency will be required. But administrative clustering of machines according to subgroups of the organization will be natural, and cooperation/transparency within these clusters will usually be a primary issue. Authorization characteristics will vary across the clusters/independent machines. Organizations will change, and correspondingly, machines will be added to/deleted from clusters, and clusters/machines will be added to/deleted from networks. Distributed system designs must be prepared to cope with these configurations and changes in configuration.

Distributed Services Design Goals

The primary design goals in our design of Distributed Services were

Local/Remote Transparency in the services distributed. From both the users' perspective and the application programmer's perspective, local and remote access appear the same.

Adherence to AIX Semantics and UNIX Operating System Semantics. This is corollary to local/remote transparency: the distribution of services cannot change the semantics of the services. Existing object code should run without modification, including data base management and other code which is sensitive to file system semantics.

Remote Performance = Local Performance. This is also corollary to transparency: if remote access is noticeably more expensive, then transparency is lost. Note that caching effects can make some distributed operations *faster* than a comparable single machine operation.

Network Media Transparency. The system should be able to run on different local and wide area networks.

Mixed Administrative Environments Supported. This was discussed in the previous section. Additionally, services must be designed to make the administrator's job reasonable.

Security and Authorization Comparable to a Single Multiuser Machine.

RT/DS File System

Remote Mounts

Distributed Services uses "remote mounts" to achieve local/remote transparency. A remote mount is much like a conventional mount in the UNIX operating system, but the mounted filesystem is on a different machine than the mounted on directory. Once the remote mount is established, local and remote files appear in the same directory hierarchy, and, with minor exceptions, file system calls have the same effect regardless of whether files(directories) are local or remote[5]. Mounts, both conventional and remote, are typically made as part of system startup, and thus are established before users login. Additional remote mounts can be established during normal system operation, if desired.

Conventional mounts require that an entire file system be mounted. Distributed Services remote mounts allow mounts of subdirectories and individual files of a remote filesystem over a local directory or file, respectively. File granularity mounts are useful in configuring a single system image. For example, a shared copy of `/etc/passwd` may be mounted over a local `/etc/passwd` without hiding other, machine specific, files in the `/etc` directory. Directory granularity and file granularity mounts are now also allowed with AIX local mounts.

Distributed Services does not require a file system server to export/advertise a file system before it can be mounted. If a machine can name a directory/file to be mounted (naming it by node and path within that node), then the machine can mount the directory/file if it has the proper permissions. The essential permission constraints are

1. Superuser (root) can issue any mount.
2. System group[6] can issue local device mounts defined in the profile `/etc/filesystems`.

[5] The traditional prohibition of links across devices applies to remote mounts. In addition, Distributed Services does not support direct access to remote special files (devices) and the remote mapping of data files using the AIX extensions to the `shmat()` system call. Note that program licenses may not allow execution of a remotely stored copy of a program.

[6] In AIX, we have given the system group (gid 0) most of the privileges traditionally restricted to the superuser. Only especially "dangerous" or "sensitive" operations are restricted to the superuser [Loucks 1986].

3. Other users/groups are allowed to perform remote directory/file mounts[7] if the process has search permission for the requested directory/file, owns the mounted upon object (directory/file) and has write permission in the parent directory of the mounted upon object.

The objectives of these constraints are to maintain system integrity but allowing users the flexibility to perform "casual" mounts. Userid/groupid translation, as discussed below, is implicit in the above definitions.

File System Implementation Issues

Virtual File Systems. The Distributed Services remote mount design uses the Virtual File System approach used with NFS [Sun 1986]. This approach allows construction of essentially arbitrary mount hierarchies, including mounting a local object over a remote object, mounting a remote object over a remote object, mounting an object more than once within the same hierarchy, mount hierarchies spanning more than one machine, etc. The main constraint is that mounts are only effective in the machine performing the mount.

Inherited mounts. It is desirable for one machine to be able to "inherit" mounts performed by other machines. For example, if a machine has mounted over `/usr/src/icon` and a second machine then mounts the first machine's `/usr/src`, it might be desired that the second machine see the mounted version of `/usr/src/icon`. This would not happen in the default case, but Distributed Services provides a query facility as part of a new `mntctl()` system call. The `mount` command supports a `-i` (inherited) flag which causes the query to be performed and the additional mounts to be made. By use of inherited mounts, clients of a file server need not know of restructuring of the server's mounts underneath the initial mount. For example, if a client always uses an inherited mount of `/usr/src`, it does not need to change it's configuration files when the server uses additional mounts to provide the subdirectories of `/usr/src`.

lookup. In conjunction with using the Virtual File System concept, we necessarily have replaced the traditional `namei()` kernel function, which translated a full path name to an i-number, with a component by component `lookup()` function. For file granularity mounts, the string form of the file name is used, along with the file handle of the (real) parent directory. This alternative to using the file handle for the mounted file allows replacement of the mounted file with a new version without loss of access to the file (with that name). (For example, when `/etc/passwd` is mounted and the `passwd` command is used, the old file is renamed `opasswd` and a new `passwd` file is produced. If we used a file handle for the file granularity mount, then the client would continue to access the old version of the file. Our approach gives the, presumably intended, effect that the client sees the new version of the file.)

Statelessness and Statefulness. One of the key implementation issues is the approach to "statelessness" and "statefulness." Wherever it is practical to use a stateless approach, we have done so. For example, our remote mounts are stateless. However, in some areas where we believe a stateful approach is necessary, we maintain state between server and client and are prepared to clean up this state information when a client or server fails. In particular, we maintain state with regard to directory and data caching, so that cache consistency can be assured.

Directory Caching. Use of component by component lookup means, in the worst case, that there will be a `lookup()` remote procedure call for each component of the path. To avoid this overhead in typical path searches, the results of `lookup()` calls are cached in kernel memory, for directory components only. Cached results may become invalid because of directory changes in the server. We believe that state information must be maintained for purposes of cache validity. Whenever any directory in a server is changed, client directory caches are purged. Only clients performing a

[7] Remote device mounts are not supported, but the only practical effect is that a remote device that is not mounted at all at the owning machine can not be remote mounted. This is likely desirable, since this situation is only likely to occur during maintenance of the unmounted device.

lookup() since the previous directory change are notified, and they, of course, only purge the entries for the server that had the directory change. This purpose of this strategy is to keep the directory cache entries correct, with little network traffic.

Data Caching. Distributed Services uses data caching in both client and server, to avoid unnecessary network traffic and associated delays. The caching achieves the traditional read ahead, write behind and reuse benefits associated with the kernel buffer cache, but with both client and server caches. As a result, read ahead (write behind) can be occurring in the client cache with regard to the network and in the server cache with regard to the disk. *As a result, disk to disk transfer rates to/from remote machines can be substantially greater than local rates.* In AIX we have carefully tuned the local disk subsystem, yet use of cp for remote files yields significantly higher disk to disk throughput than for local only files. Note that stateless designs may not support write behind, in order to guarantee that all data will be actually on the server's disk before the write rpc returns to the client.

Data Cache Consistency. In general, it is difficult to keep multiple cached data blocks consistent. We designed a general cache invalidation scheme, but chose to implement instead a state machine based on current opens of a given file. We emphasize that this mechanism is applied at a file granularity, and that it is strictly a performance optimization – the mechanism is designed to preserve the traditional multireader/multiwriter semantics of the UNIX file system. Any particular file will be in one of the following states:

1. Not open.
2. Open only on one machine This may be a different machine than the server for the file. ("async mode")
3. Open only for reads on more than one machine. ("read only mode")
4. Open on multiple machines, with at least one open for writing. ("fullsync mode")

We believe that the read only and async modes are dominant in actual system operation, and our client caching applies to these modes only. In fullsync mode, there is no client caching for the given file, but the server caches as in a standalone system.

Close/Reopen Optimization. A frequent scenario is that a file is closed, say by an editor, and then immediately reopened, say by a compiler. Our data cache consistency mechanisms are extended to allow reuse of cached data blocks in the client data cache, if and only if the file is not modified elsewhere between the close and subsequent reopen.

Kernel Structured Using Sun "vnode" Definition. We have used the Sun vnode data structure [Kleinman 1986] to support multiple file system types in the AIX kernel. This allows a clean division between the local AIX filesystem code and the remote filesystem code.

Virtual Circuit Interface. Distributed Services assumes virtual circuits are available for network traffic. One or more virtual circuits must remain in force between a client with a file open and the server for that file. (The mere existence of a remote mount does not require retention of a virtual circuit.) Execution of cleanup code, e.g., decrementing usage counts on open files, will be triggered by loss of a virtual circuit. The architecture of Distributed Services includes a *Virtual Circuit Interface* (VCI) layer to isolate the Distributed Services code from the supporting network code. Our current code uses the SNA LU 6.2 protocol to provide virtual circuit support, but, potentially, another connection oriented protocol, e.g., TCP, could be used. The basic primitives of the VCI are the dsrpc(), dsrpc_got() and dsgetdata() functions. dsrpc() acquires a connection with a specified machine and then issues dsrpc_got() to invoke a function on that machine. dsrpc_got() is called directly if the caller has a previously established connection available. Both of these calls return without waiting for the result of the remote function, allowing continued execution on the calling machine. dsgetdata() is used to request the result of a remote functions; it will wait until the result is available.

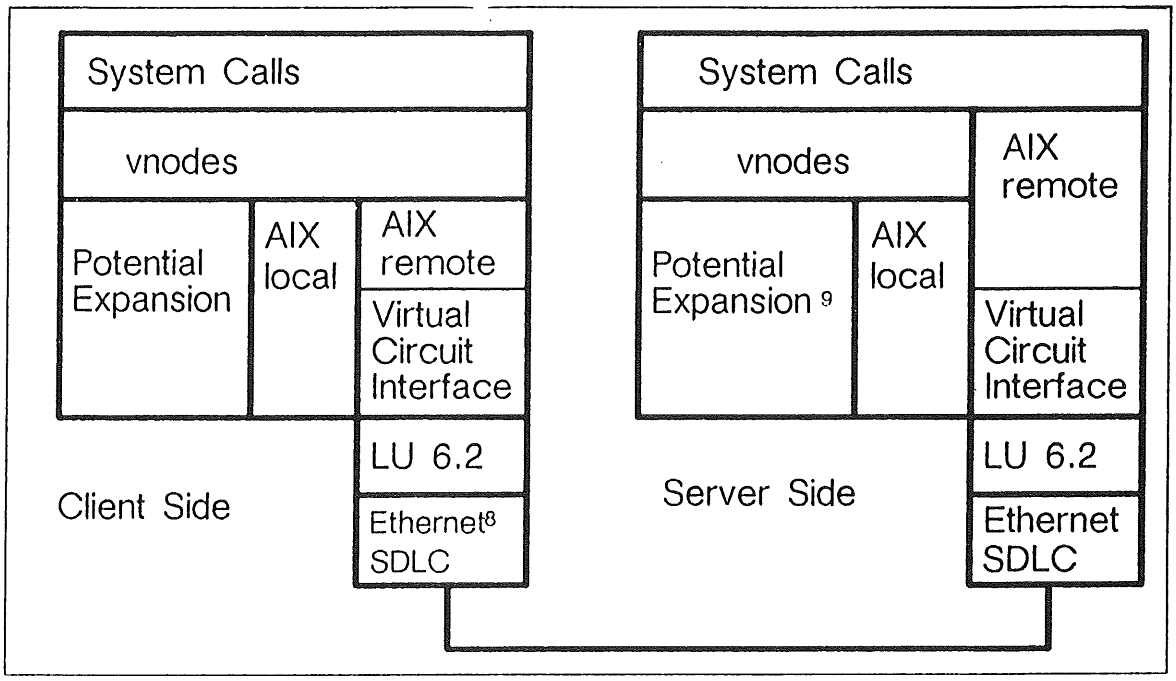


Figure 2. Architectural Structure of Distributed Services File System

SNA LU 6.2 Usage. We chose to use LU 6.2 because of its popular position in IBM's networking products and because of its technical advantages. In particular, LU 6.2 allows for "conversations" within a session. Conversations have the capabilities of virtual circuits, yet with low overhead of the order typically associated with datagrams. Typically, one or two sessions are opened to support the flow between two machines, regardless of the number of virtual circuits required. We have carefully tuned the LU 6.2 implementation, exploiting the fully preemptive process model of the AIX Virtual Resource Manager [Lang, Greenberg and Sauer 1986]. By properly exploiting the basic architecture of LU 6.2 and careful tuning, we have been able to achieve high performance without using special private protocols [Popek and Walker 1985] or limiting ourselves to datagrams.

The AIX implementation of LU 6.2 supports both Ethernet and SDLC transport. The AIX LU 6.2 and TCP/IP implementations are designed to coexist on the same Ethernet – in our development environment, we use both protocols on a single Ethernet, e.g., TCP for Telnet and/or X Windows and LU 6.2 for Distributed Services.

Distributed Services Security and Authorization

Encrypted Node Identification

When considering networks of the sort suggested by Figure 1, it is clear that each machine needs to be suspicious of the other machines. If a machine is going to act as a server for another, it should have a mechanism to determine that the potential client is not masquerading. The AIX implementation of SNA LU 6.2 provides an option for encrypted node identification between a pair of communicating machines. The identification is by exchange of DES encrypted messages. The identification occurs at session establishment time and at random intervals thereafter. Once a

[8] Ethernet is a trademark of Xerox Corporation.

[9] This is not intended as speculation of future products.

client/server have each determined that the other is not masquerading, then they can take appropriate actions authorized according to (the translated) userid's/groupid's associated with each request.

Userid/Groupid Translation

There are a number of reasons why a common userid space and a common group id space are impractical in the environment of Figure 1:

1. An individual machine, whether a private machine or a server, should not be required to give superuser (root) authority to a request from a process with root authority on another machine. Rather, it should be possible to reduce the authority of the remote process. The reduced authority may retain some administrative privileges, may be that of an ordinary user or may be no access at all, depending on the preferences of the administrator of the individual machine. Similar statements apply to the cluster of machines.
2. A user may have logins provided by several different administrators on several different machines/clusters, and these will typically have different numeric userids. When that user uses different machines, he/she should have access to his/her authorized resources on all machines in the network.
3. Previously operating machines may join a network or move to a new network, and existing networks may merge. When this happens, there may be different users/groups with the same numeric ids. Such reconfiguration should be possible without requiring users/groups to change numeric ids or changing userids/groupids in all of the inodes.

Our response to these requirements is to define a network wide ("wire") space of 32 bit userids and groupids. Each request leaving a machine has the userid translated to the wire userid and each request entering a machine has the wire userid translated to a local userid. The above requirements are met by proper management of the translations.

Distributed Services Administration

In addition to the normal system profiles, e.g., `/etc/filesystems`, there are profiles for both the SNA support and for Distributed Services. With these new profiles, we have taken care to organize the directories containing the profiles so that we can use remote mounts to administer remote machines, without use of remote login (or roller skates). For Distributed Services, there are three profiles, for machine ids and passwords, for userid/groupid translation and for registry of message queues.

Part of the AIX design is provision of a user interface architecture for a screen oriented ("menu") interface, to simplify system management and usage [Kilpatrick and Green 1986, Murphy and Verburg 1986]. Configuration of both SNA and Distributed Services, i.e., management of the SNA and Distributed Services profiles, is normally performed using menus conforming to this user interface architecture.

Distributed Services "Single System Image"

Our definition of "Single System Image" is as follows: *Users of the given system, users of external systems which communicate with the given system and application programmers ARE NOT aware of differences between single and multiple machine implementation. System administrators and maintenance personnel ARE aware of distinctions amongst machines.*

User/Programmer View of Distributed Services Single System Image

Though there are inherent exceptions to this, e.g., the `uname()` system call is designed to return the machine name, we believe that Distributed Services largely meets this definition. The key mechanism is to be able to properly configure the several machines so that they share the files and directories which matter to the user and the application programmer. These include basic profiles such as `/etc/passwd`, home directories, and directories containing applications, commands and libraries. Figure 3 sketches one such possible configuration.

Once this is accomplished, most of the desired properties just fall in place. The login process will be the same because of the sharing of `/etc/passwd` related files. Normal file system manipulations and applications work in the shared directories. Administrative commands for ordinary users, e.g., `passwd`, also work properly if they follow reasonable conventions (we had to rework several commands such as `passwd`, as discussed below.)

Administrator's View of Single System Image Configurations

Some of the administrator's tasks must be be performed for each machine individually. For example, the administrator must install and configure AIX and Distributed Services on each machine. Other tasks can be performed once for the entire single system image cluster. For example, installation of an application, in the usual case where the `installp` command retrieves files from diskette and places them in the appropriate subdirectory of `/usr/lpp`, need only be done once, assuming it is done after normal system startup. Similarly, the `adduser` command, which creates an entry in `/etc/passwd`, creates a home directory and copies standard files to the home directory, need only be applied once.

Routine maintenance, e.g., backing up and restoring files, can be done for the system as a whole while the system is in normal operation. Error logs are intentionally kept separately for each machine - otherwise, the first problem determination step would be to isolate the anomalous machine. Some maintenance operations, e.g., image backups of disks and hardware diagnostics, are necessarily performed on a machine by machine basis, while the machine is in maintenance mode.

Implementation Issues in Distributed Services Single System Image

There is an obvious question of ordering in starting the separate machines. We have added a number of options to the `mount` command and `/etc/filesystems` to allow simple retry mechanisms to be executed in the background when initial mount attempts fail. This is done to allow arbitrary ordering of the startup of machines.

Many of the interesting commands, e.g., `passwd`, use private locking mechanisms, e.g., based on creating/deleting dummy lock files. We have had to modify a number of these commands to use the `lockf()` system call.

A more subtle issue is the "copy/modify/unlink/relink" idiom used in a number of interesting programs such as editors. This idiom does not work in all cases of file granularity mounts, because a client may be attempting to violate the prohibition of linking across devices. In more detail, the idiom is as follows, for updating `foo` in the current directory:

1. `cp foo .foo.tmp`
2. modify `.foo.tmp`
3. `rm foo`
4. `ln .foo.tmp foo`
5. `rm .foo.tmp`

If `foo` is a file mount from a different device, step 4 will fail. We have had to modify several programs to do a copy if the link step (4) fails. Note that this is not a problem with directory mounts, only file granularity mounts.

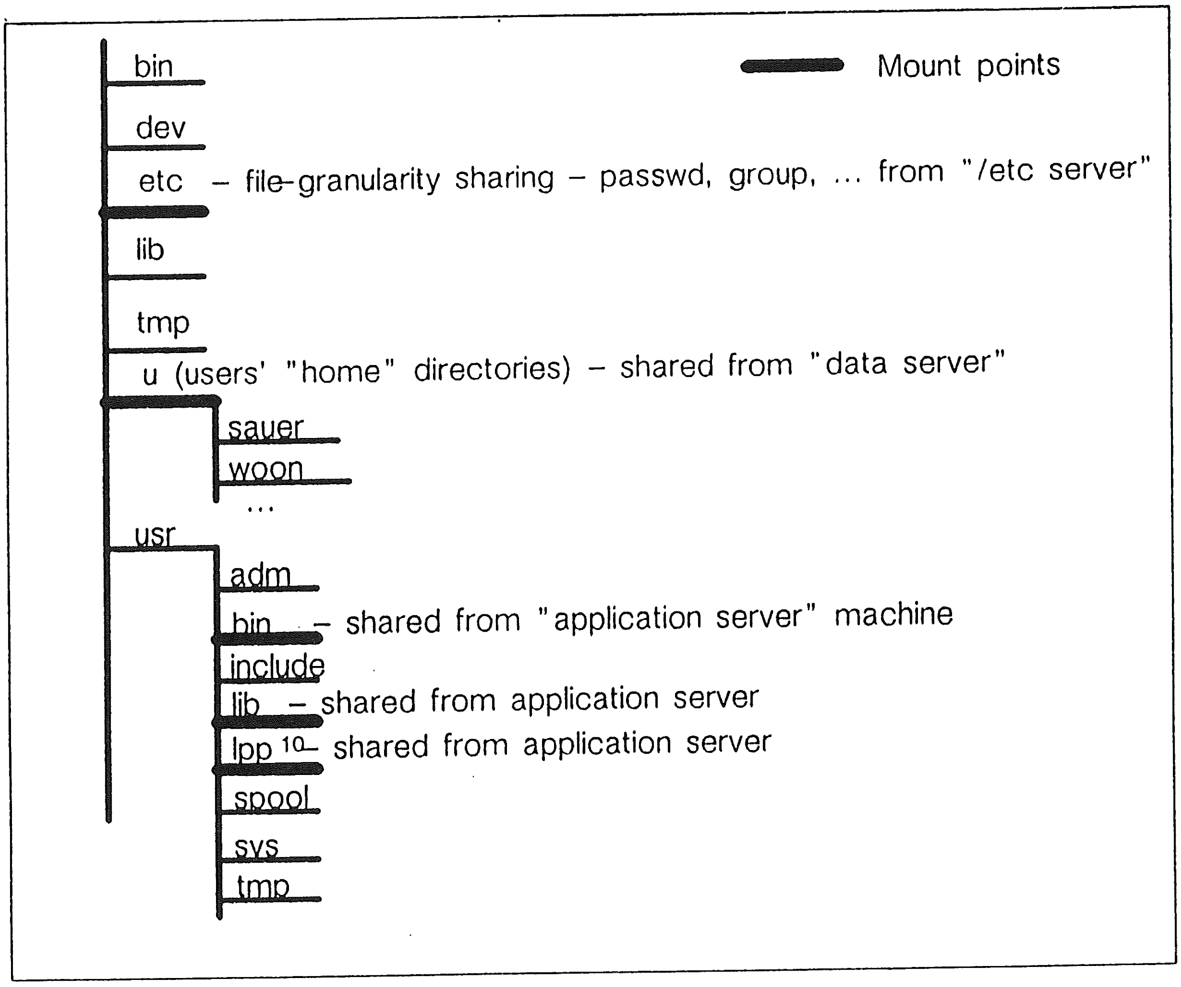

Mount points
bin
dev
etc – file-granularity sharing – passwd, group, ... from "/etc server"
lib
tmp
u (users' "home" directories) – shared from "data server"
sauer
woon
...
usr
adm
bin – shared from "application server" machine
include
lib – shared from application server
lpp 10– shared from application server
spool
sys
tmp

Figure 3. Example Shared File System.

There is also a potential problem with routines such as `mktemp()` and `tempnam()`, which use process ids to generate unique file names. Since process ids are not unique across machines, we have modified these routines to use the machine id as well as the process id in deriving a file name. (The modified versions of these routines are packaged with AIX, so that object code does not have to be recompiled/relinked to run with Distributed Services.)

Separate Machine Operation

Clearly, it is desirable that a client machine of the servers in Figure 3 be able to operate if one or more of the servers is down. A critical aspect of this is having recent copies of the shared files from the "`/etc` server." As part of the mounting of these files, before the mount is actually performed, the file is copied from the server to the client. For example, before mounting the shared `/etc/passwd` over the client `/etc/passwd`, the shared version is mounted temporarily over another file and copied to `/etc/passwd`. For each user that is to be able to use a machine when the "home directory server" is not available, a home directory must be created and stocked with essential data files. Similarly, for a machine to be able to use an application when the "application server" is not available, that application must be installed in the client's `/usr/lpp`, when the server's `/usr/lpp` is

[10] An AIX convention is to place most applications in subdirectories of `/usr/lpp`.

not mounted. The resulting machine is certainly not as useful as when the servers are available, but it is usable, and much better than no machine at all.

Summary

We believe we have done well in meeting our design goals:

1. Distributed Services provides local/remote transparency for ordinary files (both data and programs), for directories and for message queues.
2. Our implementation adheres closely to AIX semantics, except for the lack of support for remote mapped files.
3. We have achieved good remote performance in general, and some remote operations are actually faster than corresponding local operations.
4. Use of a popular network protocol, SNA LU 6.2, gives us synergy with other SNA development and independence of the underlying transport media.
5. We have been careful to provide for flexibility in configurations and administrative environments.
6. Our encrypted node identification and id translation mechanisms give us strong control over security and authorization.
7. Our use of architectures such as LU 6.2, the vnode concept, our Virtual Circuit Interface, etc. allows us substantial room for potential extension and growth in network media, file systems and network protocols, respectively.

Further, we believe we have advanced the state of the art with the following

1. Our simple, but effective approach to single system image.
2. Use of a standard virtual circuit protocol, SNA LU 6.2, while achieving high performance.
3. Our performance optimizations, especially our caching strategies.
4. Our extensions for administrative flexibility and control, e.g., file granularity mounts, inherited mounts, administration based on remote mounting of profiles, etc.

References

1. IBM, *IBM RT Personal Computer AIX Operating System Technical Reference Manual*, SA23-0806, January 1986.
2. P. J. Kilpatrick and C. Greene, "Restructuring the AIX User Interface," *IBM RT Personal Computer Technology*, SA23-1057, January 1986.
3. S. R. Kleinman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *USENIX Conference Proceedings*, Atlanta, June 1986.
4. T. G. Lang, M. S. Greenberg and C. H. Sauer, "The Virtual Resource Manager," *IBM RT Personal Computer Technology*, SA23-1057, January 1986.
5. L. K. Loucks, "IBM RT PC AIX Kernel - Modifications and Extensions," *IBM RT Personal Computer Technology*, SA23-1057, January 1986.
6. T. Murphy and R. Verburg, "Extendable High-Level AIX User Interface," *IBM RT Personal Computer Technology*, SA23-1057, January 1986.
7. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin and G. Thiel, "A Network Transparent, High Reliability Distributed System," *Proceedings of the 8th Symposium on Operating Systems Principles*, Pacific Grove, CA, 1981.

8. G. Popek and B. Walker, *The LOCUS Distributed Operating System,* MIT Press, 1985.

9. A. P. Rifkin, M. P. Forbes, Richard L. Hamilton, M. Sabrio, S. Shah and K. Yueh, "RFS Architectural Overview," *USENIX Conference Proceedings,* Atlanta, June 1986.

10. R. Sandberg, D. Goldberg, S. Kleinman, Dan Walsh and B. Lyon, "Design and Implementation of the Sun Network File System," *USENIX Conference Proceedings,* Portland, June 1985.

11. Sun Microsystems, Inc., *Networking on the Sun Workstation,* February 1986.

Book Reviews

UNIX System Security

by Patrick H. Wood and Stephen G. Kochan
(Hayden Book Company) 299 Pages, $34.95

Reviewed by Robert E. Van Cleef
NAS-RNS Workstation Subsystem Manager
General Electric Corporation
NASA Ames Research Center
Mail Stop 258-6
Moffet Field, CA 94035

Summary: A very valuable book. A must read for any UNIX system administrator.

With the current proliferation of UNIX systems in areas outside of the traditional University and Research environments, UNIX system security is becoming a real concern for many people. Unfortunately, most of the currently available books on the UNIX operating system avoid any discussion of security beyond using user passwords and basic file protection. Current system manuals only discuss bits and pieces of system security in a haphazard fashion. This means that anyone wanting to evaluate their system's security has a problem. In fact, even if you are an experienced computer user, if you come from a non-UNIX environment you will not be able to find the information you need in the manuals that are being delivered with many of the newer UNIX boxes, including AT&T's.

Many of us "old-timers" were first introduced to UNIX security concerns by reading D. M. Ritchie's [1] and R. H. Morris' [2] discussions on system security in Volume Two of the UNIX Programmer's Manual, but most vendors don't deliver Volume Two with their systems any more. And I wonder how many people in the small system world would have copies of Ken Thompson's Turing Award lecture, on Trusting Trust [3] in their personal libraries? Knowledge of UNIX system security can be hard to come by.

Seeing this need, Patrick Wood and Stephen Kochan have collected all of the bits and pieces into a concise and very readable reference manual that should be on the shelf of every UNIX system administrator. Their stated goal was to "teach security awareness to UNIX users and administrators." They not only achieved that goal, but they have provided a collection of tools to help the system administrator determine the level of security of their system, and to alter that level of security to meet their needs. They achieve this with a high level of readability and accuracy. I found it almost impossible to put down – highly unusual for a technical book. There was also a side affect in that some of their sample programs and shell scripts taught me some very useful "tricks."

This is not just another introduction to UNIX book. The authors assume that you understand enough about shell scripting and C programs to allow them to concentrate on their subject and not spend their time teaching programming. On the other hand, they do explain completely, and with many examples, the shell scripts and C programs that they introduce. In fact, in some ways this can be considered a "Cook Book" on system security, in that they include appendixes with the full source listings for the programs that they discuss. But it is more than just a cook book: like most good security manuals it not only shows you what security holes are in you system, but it gives you the tools to close those holes, with a complete index to allow you to find those examples when you need them.

Finally, a thought to those of you who aren't worried about security. If you don't pay attention to the security holes discussed in this book, someone else might . . .

Chapter 1, Introduction.

Chapter 2, A Perspective on Security, is the complete written testimony of Robert Morris, of AT&T's Bell Laboratory, before the House Committee on Science and Technology's subcommittee on Transportation, Aviation and Materials.

Chapter 3, Security for Users, not only covers the traditional subjects, such as password security, file permissions, Set User ID and Set Group ID programs, it also discusses the day-to-day security problems associated with such common programs as *cp, mv, ln,* and *cpio*. The discussion includes Trojan horses, viruses, and the problems with Intelligent Terminals, and how to deal with them.

Chapter 4, Security for Programmers, introduces you to the writing of secure programs. They discuss process control, file attributes, UID and GID processing, and dealing with the password file. They give you clear guidelines for SUID/SGID programs, and show you how to write a SUID/SGID program to safely allow selective access to protected data using both a shell script and a C program.

Chapter 5, Security for Administrators, starts with system file and device file permissions. In this chapter the authors put together something that I have never seen elsewhere, a full discussion of the administrator's security responsibilities, complete with a discussion on the proper use of the Super User privileges. They discuss password aging and control, and introduce the use of restricted shells. Under a summation on small system security, they also discussion the physical security of the system.

Chapter 6, Network Security, mainly discusses the UUCP world, including the Honeydanber UUCP that now is part of System V, but it does include some discussion of the problems associated with RJE links, NSC's Hyperchannel network, and AT&T's 3B Net, and using encrypted data links.

Appendix A – References.

Appendix B – Security Commands and Functions.

Appendix C – Permissions.

Appendix D – Security Auditing Program. *secure* – perform a security audit on a UNIX system.

Appendix E – File Permission Program. *perms* – check and set file permissions.

Appendix F – Password Administration Program. *pwadm* – perform password aging administration.

Appendix G – Password Expiration Program. *pwexp* – prints weeks to expiration of user's password.

Appendix H – Terminal Securing Program. *lock* – locks terminal until correct password is entered.

Appendix I – SUID/SGID Shell Execution Program. *setsh* – run a SUID/SGID/execute-only shell.

Appendix J – Restricted Environment Program. *restrict* – establishes a user in a restricted environment.

Appendix K – DES Encryption Program. *descrypt* – encode/decode using DES.

Appendix L – SUID Patent. A copy of US Patent 4,135,240, D. M. Ritchie's patent of the SUID concept.

Appendix M – Glossary

Index

References

[1] D. M. Ritchie, "On the Security of UNIX," UNIX Programmer's Manual, Section 2, AT&T Bell Laboratories.

[2] R. H. Morris and K. Thompson, "Password Security: A Case History," UNIX Programmer's Manual, Section 2, AT&T Bell Laboratories.

[3] K. Thompson, "Reflections On Trusting Trust," 1983 ACM Turing Award Lecture, CACM, Volume 27, Number 8 (August 1984), pp. 761-763.

troff typesetting for UNIX Systems

by Sandra L. Emerson and Karen Paulsell
(Prentice Hall, 1987, ISBN 0-13-930959-4)

Reviewed by Jaap Akkerhuis
CWI, Amsterdam
mcvax!nl.cwi!jaap

Goal of the Book

The book is intended to be an introduction to the use of troff for the novice and also a reference manual for experienced users. It tries to correct the lack of adequate end-user documentation for troff. Alas, any explanation about the concepts of troff -or any other formatting program is missing. For instance, the term "partial collected lines" is used a lot but never explained.

As an introduction to troff the authors explain all the basic requests and how to write macros. It is a pity that they do so in a haphazard way. They often use a request, like `.de`, with the remark that the full details will be explained further on in the book. This is sometimes confusing. Apparently, the authors did not have a clear idea on how to introduce a novice to the game of troff.

What I do like is that they give a full treatment of the `.nx` and `.rd` requests. Hardly any of the existing literature explains the possibilities of creating form letters with n/troff using these requests. Also, every possible troff request is explained, each description accompanied with an example of its use. But for the more experienced user there is not a lot new. Even small tricks, for example, what you can do with the `.ss` request, are not explained. Fancy techniques, like how to do balanced columns, are not handled at all. The chapters about the preprocessors and macro packages are sketchy and don't give more information than the existing literature.

To be a reference manual, it should at least replace the original n/troff reference manual. Some finer points haven't been covered, like the full definition of certain requests, for instance, the append to macro command: `.am xx yy`. So, don't throw the original manual from Ossanna out of the window; you will still need it.

Typesetting

The most disturbing and misleading thing about the book is its title. Apart from a remark like "You should think as a typesetter," there is nothing in the book about typesetting or the noble art of typography. All the examples deal with the standard non-interesting cases of typesetting.

The typesetting of the book itself is not really done exceptionally well, it is just another book which is typeset by the authors. I'm always wondering why authors don't ask advice from a typographical consultant, it would do miracles for book design. Of course, this is partly the failure of the publisher. These firms are more and more interested in making money by cranking out printed paper and not caring at all about how the product looks. I'm afraid that ignoring the issues involved with typography in this book will lead to even more horrible looking books than there are around already.

Errors in the Book

In general, there will always be errors in books. In this case, the advanced troff user will spot them easily, but for the novice they may be very disturbing.

The first one pops up in the first example in the first chapter (pages 3 & 4). This one can be waved away if you consider that novice shouldn't be hampered too much with details, but the next example (page 5) is unforgivable. The quoted troff source of `.PP` for the *-ms* macro package is missing some back slashes! This demonstrates again that it is not always easy to write about a tool by using it. There are more places in the book where these things happen. When showing the pitfalls of the arithmetic in troff using the `.ll` request the complete promised test file isn't around. Some parts of how the file might have looked and some of the (incorrect) output is shown. Something really went wrong there.

Who Should Buy the Book

Although I'm not very impressed by the book, it may be of some use for a lot of people. There are many UNIX systems around which don't provide the original documentation. For these cases, the book fills a gap. Also, people complaining about the terseness of the original reference manual might want to read it.

Work-in-Progress Reports from the Phoenix Conference

The Siemens RTL Tiled Window Manager

Ellis S. Cohen
Mark R. Biggers
Joseph C. Camaratta

Siemens Research & Technology Laboratories
105 College Road East
Princeton NJ 08540-6668
(609) 734-6524
siemens!ellis (uucp)
ellis.cohen@a.gp.cs.cmu.edu (arpa)

The Siemens RTL Tiled Window Manager is a network window manager which is currently implemented on a Sun and interacts with clients using the CMU/ITC Andrew protocol. It has been in operation since Fall 1986, and is the window manager of choice within our group. It is currently being ported to run as a window manager for X.

Except for menus and transient pop-up windows, a tiled window manager does not permit windows to overlap. If opening, moving, or enlarging a window would cause overlap, then either the operation is disallowed, or windows are automatically shrunk, moved, or closed to avoid the overlap.

Unlike other tiled window managers which lay out windows in columns, the Siemens RTL window manager supports arbitrary tiled layouts and thus avoids the limitations of other systems. The system is also distinguished by a number of important features including:

- Minimum sizes to better support what appear as "slivers" in overlapping systems – the portion of a window which remains partially visible while other windows are fully visible.
- Desired sizes – a preferred size of the window (as set by the user) which the system automatically attempts to maintain. A window may automatically be enlarged to its desired size when it becomes the focus – analogous to exposing a window upon focus in an overlapping system.
- Zooming as both a means of temporarily enlarging a window and icons to represent windows which are closed.
- Automatic placement for finding the best place to open a window.
- Automatic prorating for fairly allocating space when not all windows on the screen can attain their desired size as well as automatic filling for fairly allocating additional space to windows when extra space is available on the screen.
- Automatic plowing for shrinking or moving windows out of the way when a window is opened or enlarged. Plowing is used instead of prorating when the goal is to cause changes in as few adjacent windows as possible.
- Enlargement for making a window as large as possible by shrinking, but not closing, other windows on the screen.
- The use of dynamically resettable options (initialized via profiles) for controlling the degree of automation and the character of the user interface.

Camelot: A Full Function, Distributed Transaction Facility for the MACH, BSD 4.3-Compatible Operating System

Jeffrey L. Eppinger

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
(arpanet: jle@spice.cs.cmu.edu)

The Camelot distributed transaction facility is designed to simplify the construction of reliable, distributed applications that access shared data. Camelot provides simple interfaces (via macros and C library calls) for executing transactions and defining servers that encapsulate permanent data objects. Camelot runs on the Mach, BSD 4.3-compatible operating system and should execute on all the uniprocessors and multiprocessors that Mach supports. Implementations of Camelot for the RT PC and VAX computers are currently being tested. Some work remains before the system can be released to others. Josh Bloch, Dean Daniels, Rich Draves, Dan Duchamp, Sherri Menees, Alfred Spector, Dean Thompson, and the speaker have designed and built Camelot.

This presentation will briefly describe Camelot and its status. In particular, it will describe Camelot's goals, features, programming interface (i.e., man 3 library interface), implementation, and performance. Camelot will demonstrate that transaction processing is an easy to use, useful programming technology that can be very efficiently implemented for the Mach environment. A user's guide and other documentation can be obtained from the speaker.

Authentication for Untrusted Networks of Untrusted Hosts

Dan Geer

Manager of Systems Development
Project Athena - E40-342F
Massachusetts Institute of Technology
1 Amherst Street
Cambridge, Massachusetts 02139
geer@athena.mit.edu

Most conventional time-sharing systems require a prospective user to identify him or herself and to authenticate that identity before using its services. In an environment consisting of a network that connects prospective clients with services, a network service has a corresponding need to identify and authenticate its clients. When the client is a user of a time-sharing system, one approach is for the service to trust the authentication that was performed by the time-sharing system. For example, the network applications lpr and rcp provided with Berkeley 4.3 UNIX trust the user's time-sharing system to reliably authenticate its clients.

In contrast with the time-sharing system, in which a protection wall separates the operating system from its users, a workstation is under the complete control of its user, to the extent that the user can run a private version of the operating system, or even replace the machine itself. As a result, a network service cannot rely on the integrity of the workstation operating system when it (the network service) performs authentication.

The Kerberos design extends the conventional notions of authentication, authorization and accounting to the network environment with untrusted workstations. It establishes a trusted third-party service, Kerberos, that can perform authentication to the mutual satisfaction of both clients and services. The authentication approach allows for integration with authorization and accounting facilities. The resulting design is also applicable to a mixed time-sharing / network environment in which a network service is not willing to rely on the authentication performed by the client's time-sharing system.

The Kerberos system will be made available in a manner consistent with our previous release of the X-Window System. Parties with serious interest in participating in a beta-test of this software are invited to contact Athena at this time.

Threads in System V: Letting UNIX Parallel Process

J. C. Wagner
J. M. Barton

Silicon Graphics, Incorporated

With the rise of numerous multiprocessor UNIX machines, programming environments that allow easy use of the newly available power are becoming more important. At SGI, considerable effort is being placed on the design and implementation of a powerful parallel programming environment.

Among the enhancements under development are a multi-threaded execution process model, expanded shared memory features, user-level synchronizing features and matching debugging capabilities. The main goals for the multi-threaded processes is to offer independently schedulable threads with a create/destroy rate an order of magnitude faster than the typical *fork*(2) system call and context switch times significantly faster than process to process. This allows threads to be used for both classic cooperating task applications as well as parallelizing compilers. Threads share all text and data of the underlying process and have the same UNIX attributes as the underlying process - one pid, one user area, same file table, etc., implying that any instance of a thread can execute inside the kernel. Other enhancements include *ptrace* support for retrieving task state information. This is combined with a graphical debugger to allow multiple threads to be displayed and debugged simultaneously. The System V shared memory facility is enhanced to allow heap allocation out of shared segments, growable segments, and system cleanup of unused segments. Programs that need multiple threads performing system calls can then simply use *fork*(2), and use shared memory as the current data space; this gives functionality superior to many system's "shared fork."

The multi-thread work is being integrated into a stable multi-processor System V kernel. The low level process/thread management and scheduling code has has been running for 2 months. The base set of kernel synchronization primitives have been implemented. Multiple tasks in a process have been working for 1 month, current performance tests show it to be ~10× the system fork rate. The kernel hooks for accessing the state information of tasks is also in place. The continued effort involves performance enhancements, full scale user synchronization primitives, debugger integration, and inter-thread messaging.

CCMD: A Version of COMND in C

Andrew Lowry
Howard Kaye
Columbia University

CCMD is a general parsing mechanism for developing User Interfaces to programs. It is based on the functionality of TOPS-20's COMND Jsys. CCMD allows a program to parse for various field types (file names, user names, dates and times, keywords, numbers, arbitrary text, tokens, etc.). It is meant to supply a homogeneous user interface across a variety of machines and operating systems for C programs. It currently runs under System V UNIX, 4.2/4.3 BSD, Ultrix 1.2/2.0, and MS-DOS. The library defines various default actions (user settable), and allows field completion, help, file indirection, comments, etc. on a per field basis. Future plans include command line editing, command history, and ports to other operating systems (such as VMS).

CCMD is available for anonymous FTP from

[CU20B.COLUMBIA.EDU]WS:<SOURCE.CCMD>*.*

For further information, send mail to:

info-ccmd-request@cu20b.columbia.edu
seismo!columbia!cunixc!info-ccmd-request

CAP - Columbia AppleTalk Package for UNIX (4.2 BSD)

(For use with AppleTalk/Ethernet bridge)

Charlie C. Kim
User Services Group
Libraries and Center for Computing Activities
Bill Schilit
Columbia University

CAP is written under UNIX BSD 4.2 and implements a portion of Apple Computer's AppleTalk protocols. In order to use this package you need an AppleTalk/Ethernet bridge (e.g. Kinetics FastPath box).

CAP routines are structured, for the most part, the same as the Apple routines described in "Inside AppleTalk" and "Inside LaserWriter." Refer to the Apple documents and the procedure comments for a complete description of the routines and how to call them.

Bill Croft's original work in this area provided the inspiration for CAP.

Availability

Release

Distribution at present is limited to anonymous FTP from CU20B.COLUMBIA.EDU. See the Release notes in the directory US:<US.CCK.CAP.D4> in the file RELEASE for information on the current release (including the file locations). There are no current plans for any other types of distribution.

MDPIC – MacDraw to PIC Converter

Daniel Klein

Avatar Corp.
5606 Northumberland
Pittsburgh, PA 15217
412/422-0285

PIC provides users of UNIX a convenient, but somewhat difficult mechanism for drawing pictures inside of troff documents. All of PIC's commands are in "plain English," but it is usually very difficult to envision the pictoral representation of a set of words. The Macintosh, on the other hand, provides users a simple to use, easy to visualize WYSIWYG (what you see is what you get) drawing mechanism. Unfortunately, the only way to interface a Macintosh with *troff* was through physical cut and paste. No automated, electronic mechanism existed.

A new program, (MDPIC, for MacDraw to PIC) has been developed, that allows users to create intricate pictures with a Macintosh, and include them directly into a *troff* document. The MDPIC program takes a binary representation of the MacDraw picture (confusingly called PICT format), and translates this picture into UNIX PIC format. Most of the Macintosh drawing primitives are translated accurately – boxes, rounded rectangles, arcs, ovals, circles, arrows, text, and lines. Fill patterns are supported, although a 100% accurate representation is not possible (the Macintosh can draw with arbitrarily colored pens, while PIC supports only a black pen). Varying line sizes, fonts, point sizes, and dotted lines are all translated accurately. In short, anything that would ordinarily be drawn with PIC can be accurately translated from the Macintosh.

MDPIC enables creators of documents to quickly generate pictures to include in their documents using a WYSIWYG editor, then translate them to a format usable by UNIX. Since this new format is human readable, changes to the pictures may either be made in the original WYSIWYG form, or in the PIC source file that MDPIC generates. Because the Macintosh presents all of its pictures graphically, the user is spared the headache of painstakingly measuring and placing all of the lines, arcs, and boxes in a picture – a task that is required by using PIC alone.

MDPIC presently runs on the Sun, VAX, and Masscomp machines, and a future port directly to the Macintosh is planned for the future.

Directional Selection is Easy as Pie Menus!

Don Hopkins

University of Maryland
Heterogeneous Systems Laboratory
College Park, MD 20742
(301) 454-1516

Simple Simon popped a Pie Men-
-u upon the screen;
With directional selection,
all is peachy keen!

The choices of a Pie Menu are positioned in a circle around the cursor, instead of in a linear row or column. The choice regions are shaped like the slices of a pie. The cursor begins in the center of the menu, in an inactive region that makes no selection. The target areas are all adjacent to the cursor, but in a different directions.

Cursor direction defines the choice. The distance from the menu center to the cursor, because it's independent of the direction, may serve to modify the choice. The further away from the Pie Menu center the cursor is, the more precise the control of the selection is, as the Pie slice widens with distance.

With familiar menus, choices can be made without even seeing the menu, because it's the direction, not the distance, that's important. "Mousing ahead" with Pie Menus is very easy and reliable. Experienced users can make selections quickly enough that it is not actually necessary to display the menu on the screen, if the mouse clicks that would determine the selection are already in the input queue.

The circular arrangement of Pie Menu items is quite appropriate for certain tasks, such as inputing hours, minutes, seconds, angles, and directions. Choices may be placed in intuitive, mnemonic directions, with opposite choices across from each other, orthogonal pairs at right angles, and other appropriate arrangements.

Pie menus have been implemented for uwm, a window manager for X-Windows version 10, for the SunView window system, and for NeWS, Sun's extensible PostScript window system. Don Hopkins did the uwm and NeWS implementations, and Mark Weiser did the SunView implementation.

Jack Callahan has shown Pie Menus to be faster and more reliable than linear menus, in a controlled experiment using subjects with little or no mouse experience. Three types of eight-item menu task groupings were used: Pie tasks (North, NE, East, etc...), linear tasks (First, Second, Third, etc...), and unclassified tasks (Center, Bold, Italic, etc...). Subjects were presented menus in both linear and Pie formats, and told to make a certain selection from each. They were able to make selections 15% faster, with fewer errors, for all three task groupings, using Pie Menus. Ben Shneiderman gave advice on the design of the experiment, and Don Hopkins implemented it in Forth and C, on top of the X-Windows uwm.

The disadvantage of Pie Menus is that they generally take up more area on the screen than linear menus. However, the extra area does participate in the selection. The wedge-shaped choice regions do not have to end at the edge of the menu window – they may extend out to the screen edge, so that the menu window only needs to be big enough to hold the choice labels.

Proper handling of pop-up Pie Menus near the screen edge is important. The menu should idealy be centered at the point where the cursor was when the mouse button was pressed. If the menu must be moved a certain amount from its ideal location, so that it fits entirely on the screen, then the cursor should be "warped" by that same amount.

Pie Menus encompass most uses of linear menus, while introducing many more, because of their extra dimension. They can be used with various types of input devices, such as mice, touch pads, graphics tablets, joysticks, light pens, arrow keypads, and eye motion sensors. They provide a practical, intuitive, efficient way of making selections that is quick and easy to learn. And best of all, they are not proprietary, patented, or restricted in any way, so take a look and feel free!

References:

"Pies: Implementation, Evaluation, and Application of Circular Menus," By Don Hopkins, Jack Callahan, and Mark Weiser (Paper in preparation. Draft available from authors.)

"A Comparative Analysis of Pie Menu Performance," By Jack Callahan, Don Hopkins, Mark Weiser, and Ben Shneiderman (Paper in preparation. Draft available from authors.)

Worldnet – Computer Networks Worldwide

John S. Quarterman

Texas Internet Consulting
701 Brazos, Suite 500
Austin, TX 78701-3243

(512) 320-9031
jsq@longway.tic.com
uunet!longway!jsq

Computer networks extend throughout the world, have millions of users, and provide unique services. Yet the only publication that has dealt with existing networks at length on a global scale is my article "Notable Computer Networks" in the October 1986 *Communications of the ACM* [Quarterman1986] (the longest article CACM has ever published and one well-received by the readers). I propose to write a book on the same subject to include material which could not be fit into the article. Ths book will discuss existing computer networks throughout the world, their interconnections, the services they provide, their composition, their administration, their users, and their effects. The article is already 40 pages (30,000 words) in CACM, or about 60 book pages. There is at least that much more textual information available, and there will be many pages of additional tables, figures, and maps, so the book should be about 200 to 250 pages long.

Reference:

[Quarterman1986] John S. Quarterman and Josiah C. Hoskins, "Notable Computer Networks," *Communications of the ACM*, vol. 29, no. 10, pp. 932-971, Association for Computing Machinery, New York, NY, October 1986.

Multiple Programs in One UNIX Process

Don Libes

[The full report appeared on pages 7-13 of the July/August 1987 issue of ;login:.]

Minutes of the AUUG Management Committee Meeting May 15, 1987

1. The meeting opened at 14:10. Present were Robert Elz (KRE), John Lions (JL), Chris Maltby (CM), and Tim Roper (TR). Apologies were received from Chris Campbell (CC), Ken McDonell (KENJ), and Lionel Singer (LS). Also present were Nobuo Saito, Jun Murai, and Kouichi Kishida from the Japan Unix Society, and Greg Webb and Ian Waters from NSWIT, hosts of the next AUUG meeting.
2. In the absence of the chairman, John Lions was elected to take the chair.
3. The minutes of the previous meeting (September 1986) were read.
4. Corrections to the minutes:
    a. The abbreviation for The Japan Unix Society in item 14 was incorrect, it should be JUS not JUG.
5. Moved (TR, seconded CM) **That the minutes as amended be accepted.** Carried (4-0)
6. Business arising from the minutes

Item 12 The token for the previous newsletter editor is now in the hands of JL, who had offered to present it, and is now waiting for a suitable occasion.

Moved (KRE, seconded TR) **That the token be presented at the next AUUG meeting, whether the recipient is present or not.** Carried (4-0)

7. There was lengthy discussion on the possibilities of a Unix meeting in Singapore in 1988. Computerworld are interested in holding an exhibition there then, and JUS would be interested in having a meeting there about the time of the ICSE10 meeting (the week of April 11).

   We would need to contact the Singapore Users Group about arrangements.

   Participants from some countries would need travel support, IBM, AT&T, and DEC might be approached.

   We support it, Singapore is an appropriate place.

   The Secretary is to write to Computerworld, and suggest they hold their exhibition in the week immediately before or after ICSE10.

   Computerworld should run the meeting.

   We need to talk to all User Groups.

8. Greg Webb presented his proposals for the August AUUG meeting.

   A committee has been formed (5 members), and the meeting dates were available (and had been for some time). Rooms for the exhibition and lectures have been booked, though there was some concern about the capacity of the rooms, a limit on numbers, or a TV relay, or parallel streams were suggested as ways around this. Running tutorials or BOF's in parallel was suggested as one reasonable alternative.

There was some discussion about the prize to be offered to the best student paper, and possible other methods of financial assistance for attendees.

9. Moved (JL, seconded TR) **That an additional sum of $1000 be made available to subsidise travel expenses to a maximum of the minimum of $300 or return economy apex air fair for any full time student whose paper is accepted and who submits a written paper by the deadline (August 14).**

10. An amendment, moved (CM, seconded TR) **That the amounts be altered to be "at the discretion of the programme committee".** Carried (4-0).

11. Motion as amended carried (4-0).

12. Yet another discussion on whether we should have published proceedings. It was accepted as a good idea, but is not required for this conference, but should be done, sometime. If proceedings are published, they should form an issue of AUUGN, either at, or after, the conference.

13. A proposal from ACMS to professionally host the exhibition was presented.

    ACMS would require "contact lists" of potential exhibitors.

14. Moved (KRE, seconded CM) **That approval be given for alternative 2 of the ACMS proposal, and to the meeting committee to negotiate with Stephen Moore, and others, re contact lists.** Carried (4-0).

15. Some discussion on possible guest speakers was undertaken, and a short list produced.

16. The meeting adjourned at 17:00, and resumed at 17:20 after the various guests had departed.

17. Moved (KRE, seconded TR) **That all memberships from the Adelaide meeting be accepted.** Carried (4-0).

18. The meeting adjourned at 17:55 and resumed at 19:05.

19. The secretary's report was presented.

    Current membership numbers were presented, together with a details of correspondence received and sent.

    Moved (CM, seconded TR) **That the secretary's report be accepted.** Carried (4-0).

20. The treasurer presented his report.

    The treasurer congratulated the Adelaide meeting organisers, whose affairs were fully wound up in a very timely, and satisfactory fashion.

Most of AUUG's money is still in the bank.

Mastercard, Bankcard, and Visa are in the process of being arranged.

Moved (TR, seconded KRE) **That the treasurer's report be accepted.** Carried (4-0).

21. There was some discussion on possible extra benefits for institutional members. These could include access to the membership list in the form of mailing labels at cheaper rates, discounts on advertising, and discounts on extra copies of AUUGN. The latter was suggested as a possibility for all members.

    It was agreed to defer this issue for presentation in AUUGN, and extra consideration, including any representations from members.

22. Discussion on membership lists was deferred to a later meeting.

23. The membership fees for the coming financial year were set.

    Moved (JL, seconded TR) **That the rate for ordinary members be $55.** Carried (2-2, KRE, CM opposed, the chairman exercised a casting vote).

    Moved (CM, seconded KRE) **That the rate for student members be $30, the rate for Institutional members be $250, and the rate for a newsletter subscription be $55.** Carried (4-0).

    It was suggested that the fee increase be deferred for a short time, to allow new members to join at the cheaper rate. This was agreed to by all.

24. A proposal from /usr/group which resulted from contacts initiated by AUUG on the topic of forming links was discussed.

    Moved (TR, seconded CM) **That the secretary make a polite negative reply to /usr/group.** Carried (4-0).

25. NZUSUGI also suggested an agreement, in response to initiatives from AUUG.

    Moved (KRE, seconded TR) **That the NZUSUGI proposal be accepted.** Carried (4-0).

26. A proposal that AUUG import 4.3BSD manuals and proceedings from USENIX, for distribution to members was discussed. Because of licensing restrictions, AUUG could only do this through some agent organisation.

    Moved (KRE, seconded CM) **That AUUG underwrite the costs of such a venture.** Carried (4-0).

27. A paper mail letter should be sent to EUUG to follow up on the tentative agreement that we have with them.

28. The next committee meeting will be held on Wednesday August 26, at Softway.

29. Other business..

AUUGN: it was noted that AUUGN would benefit from lowering the amount of white space, and paying attention to the resulting weight.

Moved (TR, seconded JL) **That the secretary write to the AUUGN editor, expressing thanks, and pleasure with the quality of the newsletter, and also pointing out that some attention to compacting white space, and to printing size with respect to postage rates is desirable.** Carried (4-0).

30. The meeting closed at 20:45.

Minutes of the AUUG Management Committee Meeting August 26, 1987

1. The meeting opened at 14:09. Present were Piers Dick-Lauder (PL), Robert Elz (KRE), John Lions (JL), Ken McDonell (KENJ) in the chair, and Tim Roper (TR). An apology was received from Chris Maltby (CM), and for the initial part of the meeting from Chris Campbell (CC).
2. Moved (JL, seconded PL) **That the meeting take place, despite the lack of formal notice.** Carried (5-0).
3. Moved (TR, seconded PL) **That the agenda as presented be ratified.** Carried (5-0).
4. The minutes of the previous meeting (May 1987) were read.
5. Corrections to the minutes:
    a. Moved (PL, seconded JL) **That the wording of item 13 be changed by altering the words "to professionally host" to "to manage".** Carried (5-0).
6. Moved (JL, seconded TR) **That the minutes as amended be accepted as an accurate record.** Carried (3-0, KENJ, PL, abstaining)
7. Business arising from the minutes

Item 6 The plaque will be presented at the dinner, at which Peter Ivanov will be present. JL to make the presentation.

Item 7 No action taken, president to ring Stephen Moore or Alan Power in the next few days to discover the current status.

Items 8—11 Information from the Programme Committee is needed to determine if any bursaries are to be presented, and if so, to present them at the dinner.

Item 20 Mastercard and Bankcard are now done, and AUUG can accept these credit cards for payments. Paperwork for Visa is proceeding.

Item 21 Nothing has appeared in AUUGN yet. Further discussion was deferred to the agenda item on membership services.

Item 23 The new rate is now in effect. It was noted that the dissenting votes in the motion on this point were proposing a higher fee, rather than no increase at all.

Item 24 No action yet taken. Clarification was sought on the nature of the proposal being rejected. The secretary explained (for members not at the previous meeting, and for the record) that the /usr/group proposal amounted to a scheme where AUUG members would join /usr/group at the normal rate, and with AUUG then obtaining some percentage of the /usr/group membership fee.

Item 25 Letter still needs to be written.

Item 26 Done. Order has been placed, delivery is imminent. After this, a second offer should be circulated to members in AUUGN (and news) for a possible second order.

Item 29 Not yet done.

— Moved (PL, no seconder) **That the minutes of meetings be circulated within two weeks of the meeting.** Discussion of this was deferred, and forgotten.

— The question of which issue of AUUGN the minutes should be published in was also raised (that is, as soon as possible, or after circulation, or after ratification at the next meeting).

8. The president had no report to present, as he had been away for most of the period since his last report. Matters of business were to be covered in other agenda items.
9. The meeting adjourned at 14:51 and resumed at 14:53.
10. The secretary presented a very brief report, including copies of some correspondence.
11. There was no treasurer's report, since there was no treasurer present. It was noted that the most recent estimate was that AUUG had about $25,000 in the bank.
12. The constitutional amendments needed for incorporation were approved at the referendum, and an application for incorporation under the *Associations Incorporation Act, 1981, Vic.* The secretary is to attempt to determine the current state of this before the AGM.
13. CC arrived, and apologised for being late, at 15:02.
14. Meeting policy guidelines: Questions to be answered included
    - method of charging for the dinner, unbundling, and FBT.
    - the equipment exhibition, to be done again?
    - credit card facilities
    - proceedings
    - student papers and bursaries
    - coordination between management committee, the hosts, and the programme committee
15. There is a meeting guidelines document, that is now out of date (again out of date). PL offered to update it and publish it in AUUGN. It should include mention of the student paper prize, and bursary.

16. Moved (JL, seconded TR) **That Piers Lauder shall prepare a document to be circulated four weeks prior to the next management committee meeting.** Carried (6-0).

17. On the issue of dinner costs FBT was decided to be not an issue.

18. However, the management committee is in favour of unbundling the cost of the dinner, PL to include this in the guidelines.

19. If the equipment exhibition is to be done again in a similar manner, it will affect the choice of a venue in Melbourne.

20. Committee members are to seek opinions from attendees and from exhibitors (independently from the organiser).

21. PL and KENJ to contact exhibitors.

22. Exhibitors should be asked if they would be willing to provide sponsorship at a future AUUG meeting.

23. Credit card facilities should be available for registrations.

24. Proceedings: there is still an issue of whether it is possible to produce proceedings to be available at the meeting. The committee feels that this would obviously be desirable, and in general there meetings should aim to do it, but it is not to be regarded as mandatory.

25. Finance: Last year's statement, and a current statement are needed before anything else can be done. This is needed very quickly, by the end of September was agreed as necessary.

26. Moved (KENJ, seconded KRE) **That the treasurer forward to the president the 1986/1987 statement, and a current financial statement, no later than the end of September, 1987.** Carried (6-0).

27. The secretary is to obtain the membership list, and all support, and move it to Melbourne in the immediate future.

28. The Post Office is to be asked to redirect AUUG's Post Office Box to an appropriate address in Melbourne.

29. Operation of the management committee: Minutes need to be prepared and circulated quickly. However, for AUUGN it was felt that a report of the meeting, rather than the formal minutes, was a more suitable procedure.

30. Moved (JL, seconded PL) **That the newsletter editor be invited to management committee meetings to prepare a report of the meeting for AUUGN, and his expenses are to be paid, if necessary.** Carried (6-0).

31. Some secretarial/bookkeeping assistance is needed.

32. There was discussion of future AUUG meetings, dates, and potential invited speakers.

33. It was decided to hold a meeting in Melbourne in February, on a standards theme, with the Winter 1988 meeting possibly in Newcastle.

34. A venue and date for the Melbourne meeting are yet to be decided, advice from the exhibition organiser should be sought.

35. There was some discussion of the possibilities of AUUG participating in some way in the various standards efforts. No conclusion was reached.

36. A possible service that might be offered to members is the distribution of software tapes from usenix. Usenix is willing to make these tapes available to AUUG at no cost. AUUG needs to determine the costs of tape duplication and distribution, KENJ to follow up.

37. The meeting adjourned at 16:17 for a conference with the meeting committee, and resumed at 17:00.

38. The secretary reported that he had contacted the solicitors who are handling the incorporation, and they had informed him that there were some more papers to sign before incorporation could proceed, but that no other obstacles were currently known. The secretary will proceed with this next week.

39. More discussion on conference organisation took place.

40. Moved (CC, seconded PL) **That the president should enter into negotiations and discussions with Wael Fada (ACMS) with a view to having his company run the next meeting in Melbourne.** Carried (6-0).

41. On links with other user groups: The position of Australian chapters needs to be tidied up. We need to rationalise the state of external links.

42. There was much discussion on the possibilities for network support. Should, or can, AUUG help, or help find organisations willing to provide support. There was general agreement that ways to help fund ACSnet are needed, but it was not clear how this should be done.

43. It was suggested that vendors be invited, or requested, to run their courses, or symposia, at about the time of the AUUG meetings. This would allow participants at their meetings to attend AUUG meetings, with mutual benefits.

44. It was recognised that AUUG's constitution needs some changes (such as creation of the office of vice president). A list of potential changes should be circulated four weeks before the next meeting. TR to coordinate.

45. It was pointed out (again) that secretarial services are needed.

46. The next management committee meeting is to be held in Melbourne on Wednesday December 9, at a time to be decided.

47. The meeting closed at 18:03.

Minutes of the AUUG Annual General Meeting August 27, 1987

1. The meeting opened at 17:35. Present were an undetermined number of members of the AUUG, and several others. The AUUG president, secretary, and committee members Campbell and Lions (and Roper??) were present. The treasurer and returning officer sent apologies (the former being overseas on business, the latter's wife having just given birth).
2. Moved (John Carey, seconded Peter Tyres) **That the required notice of the meeting be waived, and that the agenda as presented be accepted.** Carried without dissent.
3. Moved (Burn Alting, seconded Chris Campbell) **That the minutes of the AGM as published in AUUGN V7 # 2 be accepted.** Carried without dissent.
4. Moved (Peter McMahon, seconded John Carey) **That the minutes of the previous general meeting, as published in AUUGN V7 # 6 be accepted.** Carried without dissent.
5. There was no business arising from the minutes.
6. In the absence of the returning officer (John O'Brien) the president presented the returning officer's report. He announced the results of the election recently held for the AUUG management committee. Elected unopposed were McDonell (president) Elz (secretary) and Maltby (treasurer). Elected to the committee were Campbell, Dick-Lauder, Lions, and Roper. O'Brien was elected returning officer, the post of assistant returning officer is vacant. All the referenda were passed with the necessary majorities, and the modified constitution now applies.
7. The president (Ken McDonell) gave his report:
    a. Incorporation of AUUG is proceeding, final administrative details are being attended to, and it is anticipated that AUUG will be incorporated in the very near future.

    b. The newsletter is now being published regularly, and the issue numbering has now (after the issue currently at the printers) caught up. Thanks should go to the editor. *This resulted in acclamation from the audience.*

    c. The president then gave some indication on what AUUG planned to do in the future (he emphasised **what** not **when**).
        - aim to increases the level of service, including affiliations with other user groups, obtaining the usenix 4.3 bsd manuals for members, and a possible distribution of the usenix software distribution tapes, at whatever it costs AUUG to distribute them.
        - convince vendors that AUUG has value to offer, and encourage vendors to assist to raise AUUG's profile in the community.

- start paying for secretarial and administrative support, to help speed up membership enquiries and general business.

8. The secretary (Robert Elz) presented a brief report. Membership numbers are remaining approximately stable, though we are gaining more institutional members. Failure of members to renew through neglect, aided by lack of renewal notices is a problem.

9. The treasurer (Chris Maltby) was not present to give a report, and no suitable substitute was available. The president advised that a financial statement and budget will be published in AUUGN before the next general meeting.

10. There was some discussion on the possibility of advertising to increase the UNIX community's knowledge of the existence of the AUUG. However it was recognised that there is unlikely to be sufficient to be gained from this to justify the high cost. It was suggested that articles prepared for publication and submitted to the computer newspapers might be a better solution. It was also noted that representatives from several of these papers had been present at the meeting.

11. An association with ACS was suggested. It was pointed out that this had been suggested before, and rejected, and that no formal arrangement was likely. Some publicity in the ACS newsletter may be beneficial though.

12. The existence of a C users group was pointed out. It was said to be a small group (about 22 members) but is growing. The possibility of some links with this group was mentioned.

13. The question of the pricing structure of the meeting was raised, especially the bundling of the conference dinner. The president reported that the management committee had decided at its committee meeting the day before that the dinner cost be unbundled in future. This was not to be treated as a rebuff to the organisers of the current meeting, who had attempted many innovations, most of which were most successful.

14. The financial strain upon some members of attending two interstate meetings a year was raised. It was suggested that perhaps an annual meeting, and more frequent local meetings would be a better structure. Not all members present were in favour of this proposal. If this were to be done, and perhaps in any case, longer meetings might be a good idea, 3 days was suggested. The possibility of holding tutorials was also raised. Input on this issue from the AUUG members was requested.

15. Meeting closed 18:07.

News Release

/usr/group
The International Network of UNIX Users
4655 Old Ironsides Drive, Suite 200
Santa Clara, CA 95054 (408) 986-8840

/usr/group has announced the publication of a booklet entitled "POSIX Explored", a technical overview of the Full-Use IEEE 1003.1 POSIX Standard specification. The 24 page document follows the earlier publication of "Your Guide to POSIX" in March 1987.

The new publication explains the Standard's history, strengths and weaknesses. It provides an overview of the changes made to the trial-use version of the 1003.1 standard, and also covers the relationship of POSIX to the proposed ANSI C Language standard.

Also detailed in "POSIX Explored" are the changes planned or being made in both AT&T's System V and in the Berkeley 4.3 based systems in order to make them conform to the POSIX specifications. The material is intended to assist software designers and implementors build conforming applications.

Both POSIX publications are available from /usr/group, 4655 Old Ironsides Drive, Suite 200, Santa Clara CA 95054, USA (Phone +1 408 986 8840). Prices are US$1 for "Your Guide to POSIX"; "POSIX Explored" is US$7 for /usr/group members and US$10 to non-members. Bulk order discounts are available.

AUUG

Membership Categories

Once again a reminder for all ''members'' of AUUG to check that you are, in fact, a member, and that you still will be for the next two months.

There are 4 membership types, plus a newsletter subscription, any of which might be just right for you.

The membership categories are:

Institutional Member
Ordinary Member
Student Member
Honorary Life Member

Institutional memberships are primarily intended for university departments, companies, etc. This is a voting membership (one vote), which receives two copies of the newsletter. Institutional members can also delegate 2 representatives to attend AUUG meetings at members rates. AUUG is also keeping track of the licence status of institutional members. If, at some future date, we are able to offer a software tape distribution service, this would be available only to institutional members, whose relevant licences can be verified.

If your institution is not an institutional member, isn't it about time it became one?

Ordinary memberships are for individuals. This is also a voting membership (one vote), which receives a single copy of the newsletter. A primary difference from Institutional Membership is that the benefits of Ordinary Membership apply to the named member only. That is, only the member can obtain discounts on attendance at AUUG meetings, etc, sending a representative isn't permitted.

Are you an AUUG member?

Student Memberships are for full time students at recognised academic institutions. This is a non voting membership which receives a single copy of the newsletter. Otherwise the benefits are as for Ordinary Members.

Honorary Life Memberships are a category that isn't relevant yet. This membership you can't apply for, you must be elected to it. What's more, you must have been a member for at least 5 years before being elected. Since AUUG is only just approaching 3 years old, there is no-one eligible for this membership category yet.

Its also possible to subscribe to the newsletter without being an AUUG member. This saves you nothing financially, that is, the subscription price is the same as the membership dues. However, it might be appropriate for libraries, etc, which simply want copies of AUUGN to help fill their shelves, and have no actual interest in the contents, or the association.

Subscriptions are also available to members who have a need for more copies of AUUGN than their membership provides.

To find out if you are currently really an AUUG member, examine the mailing label of this AUUGN. In the lower right corner you will find information about your current membership status. The first letter is your membership type code, N for regular members, S for students, and I for institutions. Then follows your membership expiration date, in the format exp=MM/YY. The remaining information is for internal use.

Check that your membership isn't about to expire (or worse, hasn't expired already). Ask your colleagues if they received this issue of AUUGN, tell them that if not, it probably means that their membership has lapsed, or perhaps, they were never a member at all! Feel free to copy the membership forms, give one to everyone that you know.

If you want to join AUUG, or renew your membership, you will find forms in this issue of AUUGN. Send the appropriate form (with remittance) to the address indicated on it, and your membership will (re-)commence.

As a service to members, AUUG has arranged to accept payments via credit card. You can use your Bankcard (within Australia only), or your Mastercard by simply completing the authorisation on the application form.

Robert Elz

AUUG Secretary.

AUUG

Application for Ordinary, or Student, Membership Australian UNIX* systems Users' Group.

*UNIX is a registered trademark of AT&T in the USA and other countries

To apply for membership of the AUUG, complete this form, and return it with payment in Australian Dollars, or credit card authorisation, to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

- Please don't send purchase orders — perhaps your purchasing department will consider this form to be an invoice.
- Foreign applicants please send a bank draft drawn on an Australian bank, or credit card authorisation, and remember to select either surface or air mail.

I, ... do hereby apply for

☐ Renewal/New* Membership of the AUUG	$55.00	
☐ Renewal/New* Student Membership	$30.00	(note certification on other side)
☐ International Surface Mail	$10.00	
☐ International Air Mail	$50.00	
Total remitted	AUD$______	(cheque, money order, credit card)

* Delete one.

I agree that this membership will be subject to the rules and by-laws of the AUUG as in force from time to time, and that this membership will run for 12 consecutive months commencing on the first day of the month following that during which this application is processed.

Date: __/__/__ Signed: ____________________

☐ Tick this box if you wish your name & address withheld from mailing lists made available to vendors.

For our mailing database - please type or print clearly:

Name: .... Phone: .... (bh)

Address: .... (ah)

....

.... Net Address: ....

....

.... *Write "Unchanged" if details have not altered and this is a renewal.*

Please charge $_____ to my ☐ Bankcard ☐ Mastercard ☐ Visa.

Account number: _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _. Expiry date: __/__.

Name on card: ______________________________ Signed: ____________________

Office use only:

Chq: bank ________ *bsb* ___-___ *a/c* ______________ *#* ______________

Date: __/__/__ *$* *CC type* ___ *V#* ____________

Who: ____________ *Member#* _________

Student Member Certification *(to be completed by a member of the academic staff)*

I, ... certify that

.... *(name)*

is a full time student at .... *(institution)*

and is expected to graduate approximately ____/____/____ .

Title: ______________________________ Signature: __________________

AUUG

Application for Institutional Membership
Australian UNIX* systems Users' Group.

*UNIX is a registered trademark of AT&T in the USA and other countries.

To apply for institutional membership of the AUUG, complete this form, and return it with payment in Australian Dollars, or credit card authorisation, to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

● Foreign applicants please send a bank draft drawn on an Australian bank, or credit card authorisation, and remember to select either surface or air mail.

.... does hereby apply for

☐	New/Renewal* Institutional Membership of AUUG	$250.00
☐	International Surface Mail	$ 20.00
☐	International Air Mail	$100.00
	Total remitted	AUD$______

(cheque, money order, credit card)

* Delete one.

I/We agree that this membership will be subject to the rules and by-laws of the AUUG as in force from time to time, and that this membership will run for 12 consecutive months commencing on the first day of the month following that during which this application is processed.

I/We understand that I/we will receive two copies of the AUUG newsletter, and may send two representatives to AUUG sponsored events at member rates, though I/we will have only one vote in AUUG elections, and other ballots as required.

Date: __/__/__ Signed: ____________________

Title: ____________________

☐ Tick this box if you wish your name & address withheld from mailing lists made available to vendors.

For our mailing database - please type or print clearly:

Administrative contact, and formal representative:

Name: .... Phone: .... (bh)

Address: .... (ah)

....

.... Net Address: ....

.... *Write "Unchanged" if details have not*

.... *altered and this is a renewal.*

Please charge $_____ to my ☐ Bankcard ☐ Mastercard ☐ Visa.

Account number: _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _. Expiry date: __/__.

Name on card: ______________________________ Signed: ____________________

Office use only: **Please complete the other side.**

Chq: bank ________ *bsb* ___-___ *a/c* ____________ *#* ____________

Date: __/__/__ *$* *CC type* ___ *V#* __________

Who: ____________ *Member#* ________

Please send newsletters to the following addresses:

Name:	....	Phone:	.... (bh)
Address:	....		.... (ah)
	....	Net Address:	....
	....		
	....		
	....		

Name:	....	Phone:	.... (bh)
Address:	....		.... (ah)
	....	Net Address:	....
	....		
	....		
	....		

Write "unchanged" if this is a renewal, and details are not to be altered.

Please indicate which Unix licences you hold, and include copies of the title and signature pages of each, if these have not been sent previously.

Note: Recent licences usally revoke earlier ones, please indicate only licences which are current, and indicate any which have been revoked since your last membership form was submitted.

Note: Most binary licensees will have a System III or System V (of one variant or another) binary licence, even if the system supplied by your vendor is based upon V7 or 4BSD. There is no such thing as a BSD binary licence, and V7 binary licences were very rare, and expensive.

☐ System V.3 source	☐ System V.3 binary
☐ System V.2 source	☐ System V.2 binary
☐ System V source	☐ System V binary
☐ System III source	☐ System III binary
☐ 4.2 or 4.3 BSD source	
☐ 4.1 BSD source	
☐ V7 source	

☐ Other *(Indicate which)* ....

AUUG

Application for Newsletter Subscription
Australian UNIX* systems Users' Group.

*UNIX is a registered trademark of AT&T in the USA and other countries

Non members who wish to apply for a subscription to the Australian UNIX systems User Group Newsletter, or members who desire additional subscriptions, should complete this form and return it to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

- Please don't send purchase orders — perhaps your purchasing department will consider this form to be an invoice.
- Foreign applicants please send a bank draft drawn on an Australian bank, or credit card authorisation, and remember to select either surface or air mail.
- Use multiple copies of this form if copies of AUUGN are to be dispatched to differing addresses.

Please *enter / renew* my subscription for the Australian UNIX systems User Group Newsletter, as follows:

Name: .... Phone: .... (bh)

Address: .... (ah)

....

Net Address: ....

....

.... *Write "Unchanged" if address has*

.... *not altered and this is a renewal.*

For each copy requested, I enclose:

☐	Subscription to AUUGN	$ 55.00
☐	International Surface Mail	$ 10.00
☐	International Air Mail	$ 50.00

Copies requested (to above address) ________

Total remitted AUD$______

(cheque, money order, credit card)

☐ Tick this box if you wish your name & address withheld from mailing lists made available to vendors.

Please charge $_____ to my ☐ Bankcard ☐ Mastercard ☐ Visa.

Account number: _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _. Expiry date: __/__.

Name on card: ______________________ Signed: ______________________

Office use only:

Chq: bank ________ *bsb* ___ - ___ *a/c* ____________ *#* ____________

Date: __/__/__ *$* *CC type* ___ *V#* __________

Who: ____________ *Subscr#* ________

AUUG

Notification of Change of Address
Australian UNIX* systems Users' Group.

*UNIX is a registered trademark of AT&T in the USA and other countries.

If you have changed your mailing address, please complete this form, and return it to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

Please allow at least 4 weeks for the change of address to take effect.

Old address (or attach a mailing label)

Name:	....	Phone:	....	(bh)
Address:	....		....	(ah)
	....	Net Address:	....	
	....			
	....			
	....			

New address (leave unaltered details blank)

Name:	....	Phone:	....	(bh)
Address:	....		....	(ah)
	....	Net Address:	....	
	....			
	....			
	....			

Office use only:

Date: ___/___/___

Who: ____________ *Memb#* _________