EUUG

papers presented at the **European UNIX® Systems User Group Autumn Meeting**

19 - 21 September 1984
Cambridge, England

E U U G

European UNIX+ Systems User Group

Proceedings EUUG Conference

Cambridge Autumn 1984

+UNIX is a trademark of AT&T Bell Laboratories

A Project Development Environment for UNIX*

Malcolm Crowe
David Mackay
George Ball
Joe Gorman
Michael Hughes
David Jenkins

Software Tools Research Group
Paisley College of Technology

1. Introduction

A software environment comprises an operating system and an integrated toolset. A software environment to support software project development must also support team activity, that is, it must be possible for team members to work on developments and extensions in their area of responsibility without inconveniencing other members of the team.

In addition it is essential that tools exist for all phases of project devlopment work, from initial designs through to maintenance. An important area, much neglected, is the provision of tools for the project management activity, and project configuration management.

This paper gives a brief overview of a Project Development Environment for UNIX (PDE) which aims to provide facilities under Unix to anable many management tasks to be performed automatically, while allowing multiple versions of files to exist in a controlled way. This work forms part of a three-year project begun in September 1983, funded under SERC's Software Technology Initiative.

2. UNIX as a Development Environment

UNIX was built to provide a programming environment at Bell Laboratories, and it contains many tools to make the programmer's task easier. The tools were deliberately kept small, with an emphasis on versatility and simplicity. Recently, however, tools of greater sophistication, to help with design and testing of software, have begun to appear, and as more and more powerful commercial programs are ported onto UNIX

*UNIX is a Trademark of Bell Laboratories.

March 17, 1985

there will soon be adequate suuport at this level for project development.

However, UNIX has always been most popular with solo programmers, such as those found in universities, and some real shortcomings for project development soon become apparent. Even the famous "Source Code Control System" (SCCS) envisages only one version of any piece of source code being on the system at any one time. Assumptions of this kind are quite unrealistic when a team of even a few people is working on a project, and it is arguable that SCCS, and the more recent Revision Control System (RCS), are merely archiving tools.

To see this, consider an everyday scene: a user is puzzling over why some piece of standard software is causing trouble. He wants to try placing some additional printf statements to help with diagnosis. Of course, he does not want (and is not allowed) to alter the system copy at this stage, so he must move a copy of the software to another part of the system, where he can work on it in peace. In a few days, he has a new version which he makes available to his friends and in no time the system administrator is faced with a series of baffling bug reports from innocent third parties. Since the happy hacker almost certainly will have left the SCCS identifiers intact, use of "what" is no help. Perhaps for this sort of reason, all UNIX source distributions envisage a manual system of configuration management for the system itself, to be the responsibility of the local guru or wizard.

Similarly, the set of file attributes is absolutely minimal, leaving most useful information about a file to be contained in a filename extension (such as ".c"), a "magic word" placed at the start of the file, or some such other ad hoc stratagem. As file types become more complex, this problem is becoming a real one, and the general issue of cross-references between files has no elegant solution.

3. Software Development Requirements

During software development, a hierarchical approach to management is essential. The project consists of several subprojects, each with various parts, and with various supporting activities: design, code, testing, maintenance, documentation, validation, etc. At any stage, some of the material may represent work in progress, some may have been "baselined" or approved with a contracting party and is therefore subject to some kind of "change control".

To represent this situation in a file system, we require that any part of the hierarchy (not just individual files) may exist in various versions, some of which may represent the structure at a particular baseline, prototype, or release (depending on the methodology being followed). Any part of the hierarchy (including individual files) must be able to refer to designs, histories, test results, bug reports/fixes, and plans. And finally, all must be subject to management supervision and control: the various attributes should be machine readable so that automated tools can produce progress and consistency reports on the project.

On the other hand, such complexity must not obtrude. The cross references must be accessible when they are needed but not otherwise visible: there should be only one directly visible version of any part of the hierarchy. The aim should be to make the system simpler and easier to use.

We believe that the Project Development Environment, described in this paper, provides a good solution to these conflicting requirements.

4. The Project Development Environment

The PDE "database" is built on top of the UNIX file system. Objects within the database are either UNIX files, known as _unstructured_ objects, or directories, known as _structured_ objects. Three main enhancements are provided:

(1) PDE object names are distinct from their corresponding UNIX file and directory names, and are not subject to the restriction on name length which is present in most versions of UNIX.

(2) Several versions of an object may co-exist, sharing the same object name. Individual versions are selected by appending a _version identifier_ to the name. A _default version_ exists which is selected if no version identifier is supplied.

(3) Objects may have associated with them an extensible set of user and system defined _attributes_ which contain such information as the type of the object, any special access controls which apply, or cross references to related objects. In some cases, the subobjects of an object are considered to inherit the value of an attribute of the object. In this case the value is said to be a _property_ of the subobjects.

Relationships between objects are represented both physically by their relative positions within the file system hierarchy, and by the use of the objects' attributes and properties.

The particular object types and relationships will vary from project to project, depending on each individual project's requirements. A project administrator will be able to define the types of, and relationships between, objects within the area of a project which comes under his control. Such definitions apply in addition to any which may already exist above this point in the hierarchy.

5. The C library

It was considered extremely important that the process of adapting the standard UNIX toolset for the PDE should be as straightforward as possible. This was achieved by implementing the facilities of version selection and attribute management by a set of low level subroutines known as the PDE Kernel, which are called from the system call subroutines in the C library.

The attribute management routines are available to all UNIX programs as standard library routines. The activities of version selection are hidden from the user, but form part of the more general function which maps a PDE object name onto that of the corresponding UNIX file or directory. The "user" in this case is a programmer writing tools for use within the PDE.

When within the PDE, the mapping function will be invoked every time an object is accessed via one of the UNIX "system calls", for example those which are used to open, close, rename, and delete files and directories. The existing system calls have been modified to do this, but take no action if the request is made from outside the PDE. Thus programs written for use in the PDE may refer to PDE objects in the same way as more traditional UNIX programs refer to files and directories. Moreover, most existing UNIX programs may be integrated with the PDE simply by incorporating the modified system calls in place of the existing ones. This is accomplished by running the link editor on the programs using the modified system calls library.

6. Version Control

In the PDE:

(1) Objects regularly have subobjects. Version control therefore applies to hierarchies.

(2) Several versions may co-exist in a system.

(3) Version control is transparent to the extent that a naive user should be given a default version of an object, without knowing that the object is under version control.

(4) The sequence of versions of an object is not necessarily linear. More usually it is tree-structured, as new versions are not always derived from the most recently created version.

To refer to a version of an object, the object's name, or base name is qualified with a version identifier, for example:

```
            A-1
         or MyFile-3a
```

There will always be a version, nominated by the administrator, which is selected if only the base name is given, and yet the object exists in multiple versions. Apart from ensuring that the version identifiers are unique, no version numbering scheme is enforced.

A version of an object may be placed under control. Thereafter, any changes to structured objects such as adding or deleting subobjects, and any updates to unstructured subobjects (ie files) are strictly controlled. An additional set of access control flags are used. These are stored as an attribute of a controlled version. They refer to the controlled access allowed on any of the subobjects below this point in the hierarchy, unless one of the subobjects itself defines controlled access privileges effective at and below its position in the hierarchy.

The controlled access features allow automatic logging of accesses in a system log file, and restriction of updates to objects, so that the existing contents are not disturbed.

Control is applied recursively to all subobjects of a structured object. Controlled objects are owned by the PDE. Any tools which are to alter them must run in a privileged mode. Such tools will examine the additional access control flags, to see if the privileged access is indeed to be allowed. An attempt to update a controlled object using a non-privileged tool will raise the standard "permission denied" error.

Allowing several versions of an object to be fully available online is potentially very wasteful of space, as it is likely that only a few of the component objects of a version will change between versions. When a new version is being derived, the user states which of the subobjects are going to be changed. Copies are made of the specified objects, and those which are to remain unchanged exist as links to the corresponding objects in the previous version, with permissions unchanged. In Unix, linking is a feature which allows the same non-directory file to appear in several directories, possibly under different names[1]

The method used for archiving is similar to that described in[2] namely the storing of the differences between successively archived versions, but is applied recursively to structured objects. This permits a complete hierarchy to be archived as a single entity. Differences in structure between successive versions are also recorded in the archive.

The archived object will continue to be listed by "ls", but any attempt to access it will fail. A special attribute records that it has been archived, and it can be reconstructed at any time.

7. The Toolset

The full set of tools available for use within the PDE may be partitioned into three subsets.

(1) The standard UNIX toolset, namely all those available outside the PDE.

(2) A small set of PDE-specific tools, known as the *privileged tools*, which perform critical actions within the PDE.

(3) Any tools which may be integrated for use in a particular project, or to support a particular methodology. Some of these may run as privileged tools.

The privileged toolset exists within the PDE to perform actions which should not be allowed to the normal user, mainly concerned with the version control facilities. They have three phases of operation - pre-execution, execution and post-execution.

The main purpose of the pre-execution phase is to ensure that the requested operation is valid with respect to the restrictions laid down by the administrator. Most commonly this would be that the user has permission to carry out the operation on the object.

The execution phase of a privileged tool is simply the application of the tool. In UNIX interaction with the user is deliberately kept to a minimum.

The purpose of the post-execution phase is to perform any functions which will indicate that the tool in question has successfully completed its work. This generally involves setting one or more attributes of the objects concerned, or making a log entry.

The functions of the pre- and post-execution phases reflect the particular management procedures being followed in the development of a project; these functions may vary from project to project. Therefore they should not be coded as part of the tools themselves.

The actions to be performed are written down as *pre-*

execution and post-execution programs. Each object contains, as an attribute value, the name of which pre-execution program (if any) is to be executed before each privileged tool, and which post-execution program is to be executed after each privileged tool.

The pre- and post-execution programs will generally be written as C Shell programs, utilising additional features added to the C Shell for attribute manipulation.

A basic set of pre- and post-execution programs is provided, which provide very elementary control facilities. The restrictions imposed by these on the operation of the privileged tools are described below, and constitute our basic environment.

There are six privileged tools.

(1) Update the attributes of a controlled object. This may only be done by the administrator.

(2) Create a new project. This tool allows a user to create a project at the top level in the file system.

(3) Make a version of an object. This tool may be applied to controlled or uncontrolled objects. Any user with read access to the object is allowed to use this tool, unless it is controlled, when the user must be authorised by the administrator.

(4) Establish a version as the default version of that object. This can only be done by the administrator at that point.

(5) Place an object under version control. This can be done only by the administrator, and only when all conditions specified in the pre-execution program have been satisfied.

(6) Archive/restore a version. This may only be done by the administrator.

It is intended that the administrator(s) for a project will extend the facilities of the basic environment by adapting the pre- and post-execution programs and adding more specialised tools, according to the requirements of their own particular areas of responsibility within the project.

8. The Project Administrator

The Project Administrator is the individual who is wholly responsible for a portion of a project. It is he who decides on what procedures are to be followed, and what standards are to be met.

The administrator of an uncontrolled object is simply the owner of the object. A controlled object is owned by the PDE, and the administrator is some nominated user. The first administrator of a controlled object will always be the person who placed the object under control. Usually this will be the previous owner.

The administrator of an object may assign another user to be the administrator for that object. Thereafter he reverts to being a normal user, with respect to the operations he can perform on that object. He may also assign another user to be a "sub-administrator" for one or more of the sub-objects which he controls. He loses his administrative powers on these sub-objects, other than the ability to remove the sub-administrator from that position.

Considerable support is given to the project administrator by the PDE in the area of configuration management. The version selection facilities of the PDE kernel and the privileged toolset allow configuration management to be applied at all stages in the development process.

The project administrator builds a plan for his part of a project by means of constructing a set of pre- and post-execution programs for the objects. These can specify any conditions which must be satisfied before the objects can proceed to the next stage of development.

As an example, objects may be assigned an attribute TYPE. This may be source code, or documentation, or testing. The connection between a source code object and a documentation object could be represented by means of another attribute, DOCUMENTATION, whose value is the name of the documentation object to be associated with the source code object. Similarly, an attribute TESTING could connect the source code object to an object which contains various approved test routines and data.

The project administrator could define that an object of type "source code" may not be controlled until it has associated with it certain documentation, and has satisfied some testing conditions. This is done by constructing a pre-execution program for the tool which places an object under control. The pre-execution program checks that if the object is of type "source code", then the DOCUMENTATION object must exist. It also checks that the TESTING object exists, and that testing has been carried out, indicated by the attribute RESULT of the testing object being set to the value 'OK'.

Such a pre-execution program is shown below:

```
set status = 1          £ Assume failure
set type = 'attr $1 TYPE'

if ( $type == 'SOURCE' ) then
  set doc = 'attr $1 DOCUMENTATION'
  set test = 'attr $1 TESTING'

£ Test for existence of documentation object

  if ( ! -e $doc ) then
    echo Documentation not present

£ Check testing has been carried out

  else if ( 'attr $test RESULT' != 'OK' ) then
    echo Testing not completed

£ Everything must be ok by now

  else
    set status = 0
endif
```

This is a simple example. The facilities provided by connecting the attribute management routines of the PDE kernel to the C Shell allow much more sophisticated processing to be done. Configuration management procedures can thus be carried out using the PDE as supplied, plus features added by the project administrator to suit his own requirements.

9. Implementation Details

The PDE kernel functions make use of a table associated with each structured object, and stored as a hidden file in the directory representing the structured object. This table contains, for each sub-object, all of the information required to do both named and default version selection of the sub-objects, and additionally stores the attributes of each sub-object. It augments the information about an object which is held in the UNIX directory.

The "mapping table" can be compared to the Common Apse Interface Set (CAIS)[3] notion of a node as a carrier of information about an object. CAIS provides specifications for a set of Ada packages designed to promote portability of Ada development tools, and other programs using the APSE. An earlier paper[4] has shown how the PDE could support this interface.

The mapping table is locked during PDE updates, and a system of timestamps prevents conflicting updates. Updates occur during system calls such as creat() or unlink() and are carried out by the PDE kernel.

If an object has versions, an extra UNIX directory is inserted which is invisible to the user. This is done for a number of reasons: (a) directories might otherwise become overlarge, (b) if the object (with all versions) may be copied or archived more easily, (c) renaming the obejct can be done in one move (if UNIX allows this).

The "real" UNIX names of PDE objects consist of anonymous strings 14 digits long. The PDE object names are not used (a) so that long object names can be allowed even if UNIX does not support long filenames, (b) to discourage people using vanilla UNIX tools for hacking the PDE.

The mapping table is in a hidden file, but is not otherwise protected: great care is taken by the PDE to ensure that the table has the same owner, group, and access modes as the directory it refers to. This can be seen as a weakness of the present implementation, but the problem is lessened by the name translation process described above.

In porting standard UNIX tools onto the PDE, the only problems encountered have been (a) programs that read directories must be modified where necessary to use standard subroutines as in 4.2bsd, (b) some programs abuse the memory allocator (both shells do this), and this can cause problems.

Experience with the current implementation has not shown any appreciable slowing down of the system. However, there is obviously some overhead associated with version selection, and any program that needs the PDE kernel is appreciably larger in size.

A tempting development, which would solve many of the small problems noted above, would be to incorporate the PDE kernel into UNIX
by creating a new layer in the operating system at executive or supervisor level, and/or altering the UNIX directory structure to include attributes. Certainly the PDE has now reached a point of stability where this could be contemplated, but UNIX itself remains in a state of flux.

10. Conclusions - A Tailored Environment

A software development "methodology" consists of: management procedures; technical methods; and software tools. It is a function of project management to define the methodology to be used for any given project. In some cases the methodology is not supported by tools, and the procedures are manual. Current thinking[5] is that the methodology should be preeminent over the tools used in it. However, many methodologies exist, and some are appropriate only for certain types of project. For example, a methodology designed for real-time software may not be suitable for developing business oriented systems.

It is therefore not feasible to expect a single software environment to support the development of more than one type of software system with the same degree of rigour. In this paper the Project Development Environment has been described which allows project management a more flexible approach, while still retaining aspects of project control.

By providing only a minimal toolset, but giving the facilities to extend this and incorporate any existing UNIX based tools, the PDE is not restricted to supporting a single software development methodology. Additionally, within a particular methodology, project management may add additional features, and restrictions, giving an environment tailored to suit the needs of individual projects.

The system as described above is ready to undergo evaluation, and is available to other participants in the Alvey Programme. Continuing development work on the PDE will be concerned with providing the project administrator with a more formal means of defining project structure, and also further investigations into the implementation of the CAIS specifications.

In addition a comparative study is being made of the PDE and other software development environments.

References

1. D M Ritchie and K Thompson, "The Unix Time Sharing System," The Bell System Technical Journal Vol. **56**(6) pp. 1905-1929 (1978).

2. W F Tichy, "Design, Implementation, and Evaluation of a Revision Control System," Proceedings of the 6th International Conference on Software Engineering, IEEE, Tokyo, (Sept. 1982).

3. US Dept of Defense and KIT/KITIA CAIS Working Group for the Ada Joint Program Office, "Draft Specification of the Common APSE Interface Set, Version 1.1," CAIS, (September 1983).

4. M K Crowe et al, "Supporting the CAIS from UNIX," Ada UK News Vol. **5**(1) pp. 48-50 (January 1984).

5. A I Wasserman and P Freeman, "Ada Methodologies: Concepts and Requirements," ACM SIGSOFT Software Engineering Notes Vol. **8**(1) pp. 33-50 (January 1983).

68000 MEMORY MANAGEMENT UNITS and the UNIX* KERNEL

Clara S. Lai

Chris Peer Johnson

UniSoft Systems

739 Allston Way

Berkeley, CA 94710

INTRODUCTION

UNIX system performance depends on many factors, one of which is the memory management unit. Porting the UNIX kernel to Motorola 68000 and 68010 based systems has given Root a unique insight into the implementation and performance of MMU designs. This talk describes the kernel's interaction with the MMU, observations and recommendations of MMU design and implementation, and a brief discussion of the memory-management schemes Root has come across.

The following terms are used in this talk. In the simplest case, a "process" is a program, "multiple processes" are concurrent programs, and "switching processes" is the transition from executing one program to executing another. A "memory management unit" is the hardware providing memory management to the system. A "context" is the hardware concept corresponding to a "process", e.g., having an MMU with 8 contexts means 8 processes can be "mapped" or "made known" to the memory management hardware at a time.

* UNIX is a trademark of AT&T Bell laboratories

KERNEL PERFORMANCE and the MMU

Before discussing how the MMU affects kernel performance, it should be pointed out that the UNIX kernel performance depends on a wide range of factors. The MMU is only one of several factors necessary for a high-performance UNIX system. Several hardware components affect system performance; these include the CPU instruction cycle speed (8, 10 or 12.5 MHz), disks and disk controllers^, serial controllers+, data transfer-rate and bus features~, as well as the number of CPU wait states introduced by the MMU per memory cycle. Other factors affecting performance less directly include the amount of memory available (which, in turn, affects the amount of swapping or paging), the amount of decision-making required in the interrupt service routine<>, file system parameters and disk organizations, and the software required to implement the MMU. All of these factors affect UNIX kernel performance, to different degrees. This talk focuses on the impact of the MMU design.

The MMU in a UNIX system provides several basic services to the kernel. If these services were implemented entirely in software, they would require tremendous overhead. An MMU makes physical memory easily accessible to many processes simultaneously with minimal system overhead.

Each time a process references a memory address, the MMU translates this logical address into a physical address. In addition to protecting user programs from each other, the MMU also protects the UNIX kernel itself from faulty user programs. With the help of a MMU, each process runs without knowledge of or hindrance from other processes, and, at the same time, possibly sharing memory with cooperating processes. The more completely these services are

---

^ e.g., a controller with DMA performs better.

+ e.g., terminal controllers that interrupt processors for each character slow the system.

~ e.g., whether the speed of the bus can keep up with the processor and memory.

<> e.g., more logic is required to implement terminal controllers with a single interrupt vector for receiver and transmitter than for those with separate interrupt vectors.

implemented in the hardware, the faster the system performs.

Memory management hardware is generally designed for speed. Most of the proprietary MMUs are well designed, resulting in fast (zero-wait-state) memory access. However, any overhead generated in programming the MMU slows system performance. Different MMU designs require vastly different amounts of management software in the kernel. The less work the kernel has to do, the better the system performs.

The UNIX kernel performs operations requiring high-frequency interactions with the memory management unit. These operations happen many times a second, practically every time the user does data read/write, at every I/O interrupt, and at every clock interrupt. The following list summarizes the areas where the UNIX kernel and the MMU interact. These areas include memory allocation and mapping, switching processes, and accessing user memory (which can be subdivided into validating a user logical address and translating it into a physical address, and reading/writing the user memory). The kernel's objective is to perform these functions as easily as possible.

- memory allocation and mapping

Every time a new process is created, the kernel allocates chunks of physical memory for the process, locates the registers where the pieces of memory are to be mapped, then maps in the physical addresses as user virtual addresses. Ideally, the MMU should not place any requirement on memory allocation, the MMU registers should be easy to locate, and the contents that need to be loaded into the registers should be evaluated simply. Conversely, the following require additional logic in the kernel to decide where to map memory: size and boundary mapping restrictions, collision of MMU locations (i.e., more than one virtual address for one or more processes mapped in a given MMU location); multiple locations where a given virtual address can be mapped; and complicated formulas for locating and evaluating MMU registers or locations. Other pitfalls for MMU designs in this area include requiring a huge amount of memory for in-memory page tables, overly restricting kernel virtual address space, or having portions of the MMU registers dedicated for a particular purpose.

- switching process

Switching from the execution of one process to another involves setting up the MMU hardware to map in the new process and then switching to it. This is typically done by loading several special MMU registers. The number of processes that can be simultaneously set up in the MMU hardware varies from one to many. The number of registers that need to be set up per process also varies. If only process can be mapped at a time, the mapping information for the new process has to be set up every time the kernel switches processes. If the MMU can map in two processes at the same time, then the kernel does not need to set up mapping information every time it switches between the two processes.

The objective is to reset registers infrequently and easily. Allowing many processes to be mapped at the same time, keeping down the number of registers that need to be reset for each process, and avoiding implementing part of the context-switching mechanism in software helps keep this operation simple. Any added complexity, such as disabling supervisor mode when switching context, specifically needing to unset previously mapped areas before re-use, clearing status, waiting for acknowledgement and response, checking for return value and errors, etc., increases overhead and should be done in hardware, if possible.

- validating a user logical address and translating it into a physical

address

The kernel constantly needs to read/write bytes, words, or blocks from and to user memory. Before actually doing the read/write, the kernel checks that the logical address referenced is within the user virtual address space. Validation is usually followed by translation into the physical address used by the kernel to access the data. For most MMUs, validation and translation simply require looking up one or two registers, or using built-in functions (as in the Motorola 68451). Requiring complex logic to find MMU location in a multiple- location scheme, and/or to interpret register contents once they are found adds to the overhead. For systems in which the kernel cannot read the MMU registers (write-only), the kernel either needs additional logic to keep track of the mapping information, or might need to actually map in the area through a scratch page and try touching it for validation.

- reading/writing user area

The amount of work the kernel has to do here can be viewed in three levels of difficulty. In the simplest situation, the kernel is a logical extension of the user (i.e., the kernel and the user are in the same virtual space). Moving information back and forth between the user and kernel itself and does not even require translating into physical address. An example of this is the Stanford MMU. In the second level of difficulty, the MMUs allow one-to-one mapping of virtual to physical memory for the kernel. The user virtual address is translated into a physical address; once this is done, the physical address, which is the same as virtual for the kernel, is used to access the area as if it is part of the kernel area. An example of this is the Motorola 68451. In the third case, the MMU does not allow mapping in all of physical memory for the kernel. The user virtual address is translated into physical address, then this physical address is mapped into the kernel virtual space before actually reading or writing it. In this last case, mapping in the user physical as kernel virtual requires additional checking to determine if information is crossing page boundaries.

TYPES of MMU

The most popular MMUs Root has implemented software for are the Stanford MMU and the Motorola 68451 MMU. Among the proprietary MMUs, the most commonly found are derivatives of the Stanford MMU. These derivatives range from those running the kernel out of a particular context to those eliminating segment tables. In this discussion, the MMUs are grouped as follows:

- Stanford MMU

- Motorola 68451 MMU

- Modified Standard MMU and Single-level Stanford-like MMU

- Other proprietary MMUs
  e.g., base and bound,
  multiple base and bound,
  a slave processor handling MMU

- no MMU|

The amount of work it takes to implement the MMUs listed above is briefly described here, to provide a sense for the magnitude of possible diversity from a generic kernel. The

| The requirements for this type of system are very different from one with an MMU; details are not discussed in this talk.

modifications to the kernel required by the different MMUs can be viewed in three categories. In the worst case, there is no memory management unit.

Since UNIX requires that processes be switched, massive modifications throughout the kernel are required for it to run with no MMU. This includes major surgery, such as discarding the swap scheduler (process 0) itself. In the best case, implementing the MMU requires changes only in the MMU-dependent section of the kernel. This is fairly straight-forward and does not require changes outside of the MMU-related code. An example of this is the Stanford MMU. Between these two categories are MMUs that require modifications of code in parts of the kernel that are not directly MMU-related, such as memory allocations. An example of this is the Motorola 68451 MMU, which requires a particular memory allocation scheme. The following is a brief discussion of the various groups of MMUs; the intention is not to describe any particular MMU in detail, but to highlight some of the characteristics related to the kernel-MMU interactions noted above.

I. The Stanford MMU

The Stanford MMU has a multi-level mapping scheme. The highest level identifies the process number, called the context. For each context, the entire virtual address space of 2 megabytes is divided into many 2k-byte pages. These pages are accessed through segments (the next level), which point to page entries (the bottom level) in a page table. Each running process has its own page tables, whose entries point to fixed-size pages. Every time a memory address is referenced, it is translated to a unique page table entry containing the physical address. For most current implementations, translation from virtual to physical address in this mapping scheme is performed so fast that the 68000 processor requires no "wait states".

One characteristic of the Stanford MMU which does not occur in most other MMUs is the presence of the UNIX kernel in the user process' virtual address space. The supervisor does not run in a special context, but is mapped into virtual addresses in the same context as the current running process, using one set of page registers which is pointed to by each context through its segment registers. Using part of the user virtual address space for the kernel slightly reduces the addressable space for a particular process, but does not pose a real problem since the reduced virtual space is still quite large*. A disadvantage of this feature is that binaries from a system with a Stanford MMU cannot be ported to systems with most other MMUs. In most other systems, the kernel does not take up any space in the user process' virtual address space, and user programs are usually origined at virtual zero. Since the kernel on a Stanford MMU system resides in low virtual memory of a user process, the process can no longer be origined at virtual zero.

II. The Motorola 68451 MMU

Unlike most MMUs which have unique direct correlation between virtual address and MMU register address, the Motorola 68451 has 32 descriptors, each of which can be used for any virtual address. Thus, a particular virtual address can be mapped using any one of the 32 descriptors; most other MMUs have a unique register where the virtual to physical information is kept. The 68451 has variable page size. Once the page size is selected, however, it restricts memory size and boundaries allocation; e.g., if a system chooses a page size of 2k, every mapping has to be of a size divisible by 2k. In addition, the boundary at which this piece of memory lies must be on a multiple of the size~. In general, a system with one Motorola 68451 is quite well suited for four to eight users, with approximately 5 segments per process.

As mentioned before, on a system with Stanford MMU, virtual to physical address can be evaluated easily because the virtual address maps into unique segment and page register. On the Motorola 68451, however, a virtual address may be mapped into any of the descriptors, so more searching is required.

The result is comparatively slower address translation for the Motorola 68451 than the Stanford MMU. A maximum of 32 different chunks of memory can be mapped at a time. Due to

---

* This reduces the addressable space from two megabytes to approximately one and a half megabytes. In UniPlus+, the kernel resides in location 0-80000 for a system with a standard Stanford MMU on 68000.

~ This creates certain problems. If one wants to map in a large process, this limits where it can be placed (e.g., if you want to map in a chunk of memory of 64k, this piece must go on a 64k boundary).

the restriction on size and boundary, the standard memory allocation scheme cannot be used, resulting in a slightly higher overhead in the kernel for memory allocation*. Memory also fragments more often under this mapping scheme.

These shortcomings are compensated for by several positive features. One advantage is fewer registers need to be set up. Special built-in functions also help to speed up several operations performed frequently by the kernel, including validating virtual addresses and translating virtual addresses into physical addresses. Despite these advantages and disadvantages, the Motorola 68451 and the Stanford MMU run similarly as far as MMU operations are concerned.

The above analysis is based on our experience with a swapping kernel; when implemented as a virtual kernel, there is a further disadvantage for the 68451. For a system with virtual memory, it is desirable to map in a lot of memory in reasonably small chunks at a time. Due to the small number of descriptors in a 68451, only a few chunks of memory can be mapped at a time. If these chunks are large, more memory can be mapped at a time, but the resolution on the page-frames is too rough for an efficient paging scheme. If a small page size is chosen, only a little memory can be mapped in at a time, and the system will page fault frequently. Compared to the 68451, the Stanford MMU scheme - with its fixed page size in small granularity - is better suited to a virtual memory system.

III. Modified Stanford MMU and Single-level Stanford-like MMU

A large group of proprietary MMUs fall into this category of variations of the Stanford architecture. A modified Stanford MMU has the same multi-level mapping as the Stanford architecture, except the supervisor runs in a separate context. A Stanford-like MMU does not have segments, but has a single-level mapping scheme with only page tables. In addition to these changes, other modifications are used in the proprietary MMUs for such things as to increase the number of processes mapped in at a time, to increase flexibility in accessing user memory, to provide faster translation of virtual to physical memory, to increase

* e.g., due to the mapping scheme restrictions, the data, the stack, and the udot are allocated separately on a system with a 68451.

protection flexibility, and to increase addressable virtual space for a process.

There are a variety of approaches to achieving additional flexibility. These include allowing more contexts, allowing larger page sizes, allowing more segments per context, using in-memory page tables, using cache memory, using a common page table for all processes, allowing variations of mapping modes, allowing fancier protection permissions, and allowing the kernel to readily access the area of any context at any time. Most of these efforts do serve their purpose, the one exception being increase in protection flexibility. Since the UNIX kernel currently only makes use of write protection for shared text and shared data, additional protection permissions are not currently used.

Some of these modifications, however, introduce new overhead in the kernel. Examples are additional logic in the kernel to resolve collisions in a common page table, to invalidate pages in cache memory when switching processes, to handle insufficient page table entries, to do additional book-keeping for switching context, and to evaluate added levels of indirection for accessing MMU registers.

IV. Other proprietary MMUs

In addition to the MMUs listed above, we have encountered other schemes. Some of these are the more conventional base and bound or multiple base and bound schemes. Others use one or a few sets of registers to do the mapping. A less conventional MMU uses a Z80 slave processor to handle all memory locations and management.

Most of these MMUs have the advantage of being simple and easy to set up by the kernel. However, some have disadvantages. Those with few registers have to be reset more often when switching processes. This might not be a problem when few processes are running, but would be more noticeable with many users. In some cases, the scheme is so simple that it is difficult to implement certain kernel features or functions requiring additional registers to map in more than the usual text, data, and stack area - such as the **phys** system call, shared text, and shared data. Severly restricting the mapping and the size of the kernel virtual space also creates a handicap when the kernel is accessing user area. For those with a huge number of registers, the kernel might require additional logic to set up only the necessary registers instead of setting all of them up when switching processes.

In general, MMUs with a limited number of registers are more suited to a swapping kernel, but are less suited to a virtual memory kernel. In the case of the slave processor MMU, which handles allocation as well as memory management, the system can turn into a virtual memory system without requiring any modifications to the kernel itself. It only requires changes in the MMU.

CONCLUSION

The amount of work done by the UNIX kernel depends on how the hardware translates virtual to physical address. A simple MMU allowing sufficient mapping registers is more desirable than a complex one requiring more effort from the kernel software. A simple design also expedites the process of porting the UNIX kernel. The overhead we have seen in a swapping kernel is expected to be more pronounced in a virtual memory system, because of the complex record-keeping and more frequent reloading of registers required. When designing a MMU, one should find a simple scheme that efficiently performs the frequent basic operations before considering fancier features for supporting virtual memory. Even though the affect of the MMU logic on system performance as a whole might not be significant, a well designed MMU does mean less work for the kernel. Using an already proven MMU instead of designing new ones, OEMs can also take advantage of the debugging and refining others have done.

Interactive Three-Dimensional Molecular Graphics under UNIX

C.Huang
T.E.Ferrin
G.S.Couch
K.C.R.C.Arnold
L.Jarvis
R.Langridge

Computer Graphics Laboratory
University of California, San Francisco
San Francisco, CA 94143, USA

ABSTRACT

The Computer Graphics Laboratory at UCSF was established in 1976 for research on the structures and interactions of proteins, DNA, drugs and other molecules of importance in biomedicine. MIDAS (Molecular Interactive Display And Simulation) is a large interactive molecular modeling graphics package developed at UCSF under UNIX, originally on a PDP 11/70 with an Evans and Sutherland Picture System 2 in black and white. It now runs on a VAX 11/750, and provides a flexible tool for the study of small and large molecules and their interactions, taking full advantage of available interactive three dimensional color display capabilities on both the Evans and Sutherland Picture System 2 and Multi Picture System and eventually on a PS300. Bond rotation, interactive monitoring of several distances and ''docking'' with real-time representations of molecular surfaces is well supported. Among its more innovative features is an unusually coherent hierarchical database for storage of macromolecules which minimizes storage space requirements and access time. The "tool building" philosophy encouraged by UNIX has resulted in a well organized and maintainable program that is suited to reimplementation on UNIX-based graphical workstations such as the Silicon Graphics IRIS. Supported by US National Institutes of Health research grant RR1081.

March 17, 1985

Interactive Three-Dimensional Molecular Graphics under UNIX

C.Huang
T.E.Ferrin
G.S.Couch
K.C.R.C.Arnold
L.Jarvis
R.Langridge

Computer Graphics Laboratory
University of California, San Francisco
San Francisco, CA 94143, USA

1. Introduction

MIDAS, which stands for Molecular Interactive Display And Simulation is a real-time, interactive, three-dimensional color graphics molecular modeling system running under UNIX*. It is the third-generation of modeling system developed at UCSF, following MMS (Molecular Modeling System), and MIDS (Molecular Interactive Display System). Molecular modeling is especially well suited for computer graphics because the graphics reduces large amounts of data to visually recognizable patterns; real-time interactive graphics is even more useful for our users because they can manipulate these patterns to discover the relationships among them[1].

2. Capabilities

The result of three years of design, programming and debugging is a user-friendly, real-time, interactive three-dimensional molecular modeling system which aids in the study and prediction of biomolecular interactions. For example, Dr. Jeffrey Blaney, while he was a graduate student at UCSF, correctly predicted the relative activities of thyroxin analogs when binding to the thyroid hormone carrier protein prealbumin[3]. In English, that means he used our software to predict how variants of a drug would interact with their receptor.

The basic capabilities of MIDAS include displaying selected atoms in large models, coloring them independently,

---

* UNIX is a trademark of Bell Laboratories.

and manipulating them via dials and joysticks. Molecular models may be displayed in full, which usually results in an incomprehensible mess; more commonly, only the backbone atoms are displayed to reduce the amount of extraneous details and produce a visually useful pattern. Color contributes greatly to legibility and also provides an additional dimension for conveying information to the observer. Being able to manipulate these patterns in real-time using the dials and joysticks is also important because it enables users to study models from different vantage points. This helps the chemist to discover geometric relations among various molecules which may not be immediately apparent from a single view. The real-time manipulation of models is most important when chemists try to understand the binding between an enzyme and a protein by ''docking'' molecular models.

Some more interesting features of MIDAS include the ability to rotate bonds, to monitor bond distances and torsional angles, to simulate atomic surfaces, and to modify molecular structure. Bond rotations are different from global manipulation of models in that they modify the structure of models instead of their position or orientation; they are important for molecular modeling because molecules are generally ''floppy'' and often change their conformations by spinning about a chemical bond. Distance and angle monitoring is done in real-time on the updated positions and orientations of the models; it aids the user when he tries to ''dock'' molecules or determine the degree of overlap between atoms. Displaying van der Waals surface† [4] of atoms provides another way of detecting the degree of overlap. The surface, represented as a set of dots, may be manipulated in the same manner as the stick model and provide a greater bandwidth of communication. Finally, the ability to modify model structure makes studying molecular interactions much easier. The user may replace pieces of a model with other predefined pieces to form similar models with user-defined properties (_e.g_. add a new group of atoms to a drug to increase the interaction between drug and receptor). The combination of these features provide a powerful tool to chemists and biochemists for interactive molecular modeling.

3. Design Constraints

MIDAS was designed primarily for modeling proteins and nucleic acids and their interactions. These molecules are

† The van der Waals surface is generated by creating a sphere around each atom according to its atomic number. The molecular van der Waals surface is the union of these spheres.

large compared to typical drug molecules: the average number of atoms in proteins and nucleic acid structures, as found in the Protein Data Bank[2] files, are 1500 and 1000 respectively, and these are only a small subset of the huge family of these molecules found in nature, the scope of which can range up into millions of atoms. The average amount of data associated with each atom is roughly 20 bytes (atom name, x, y, z coordinates, some physical properties, and graphics information). So the memory requirement for studying an average sized interaction is roughly 50 Kb, just for raw data.

We currently have two VAX 11/750's**, an Evans and Sutherland Picture System 2, an E&S Picture System 300, and a Silicon Graphics IRIS workstation. The PS2 is a high-performance vector graphics system, capable of displaying and transforming 20,000 vectors in real-time and without image flicker. The PS300 is also a vector graphics system. The IRIS workstation supports some real-time raster graphics. However, these later two graphics systems are relatively recent acquisitions; when we started designing MIDAS in 1979, the only equipment we had was the PS2 and a PDP 11/70.

The original equipment imposed some physical constraints on our design of MIDAS; the limited address space of the PDP 11/70 immediately restricts the amount of data that can be kept in memory. The computing environment imposed further constraints during the design phase of MIDAS. The 11/70 was not dedicated to just graphics. Other users did text processing, numerical calculations, and program development. We had to make MIDAS as efficient as possible. We originally had only a 300 Mb disk shared by 100 users; so we could not afford to sacrifice disk space for execution speed. Therefore, our goal when we designed MIDAS was to build a modeling system which processes a large amount of data using very little memory, gives real-time performance while consuming little CPU time, and does not use excessive amounts of disk space for storing the molecular database.

4. Implementation

The problem of a limited address space on the PDP 11/70 was solved by dividing MIDAS into three processes: one to

These numbers do not include the hydrogen atoms which are much more numerous but not important for displaying molecular structure.

** VAX, PDP, and VMS are trademarks of Digital Equipment Corporation handle the real-time interaction with the user, one to maintain the molecular database, and one to store the graphics objects so they need not be regenerated each time. Effectively, we tripled the available memory by modularizing our system. This arrangement also enforced strict modularity. The processes communicate with each other using the standard UNIX IPC mechanism of pipes. Large volumes of data are passed back and forth between processes using shared disk files. Sharing is synchronized by use of the above-mentioned pipes.

Of course, one of the modules (the database editor) would require too much memory if it were to keep all of the databases in main memory. We therefore designed a disk-based system which allows reasonably fast access time. We chose a hierarchical database format which corresponds closely in structure to the molecules we modeled. It was fairly easy to implement on UNIX because files on UNIX are byte-streams and do not have system-imposed structure. Thus, we were able to store arbitrarily sized arrays of structures in our data file with an index to each unit stored in a second file; each of the data array corresponds to a residue (a set of atoms which form a subcomponent of the overall molecule, e.g., an amino acid) in the molecule. A third file stores the connectivity information for each residue; since there is often a large number of residues of the same type (but with different atomic coordinates) in each molecule, we minimize redundancy by only storing one connectivity record for each different residue type. There are an average of 150 residues in each molecule with approximately 10 atoms per residue; so the data arrays are about 200 bytes each, and thus an average data file is about 30 Kb long. This scheme provides for a compact database arrangement with a minimum redundancy. A more complete description is given in [5]. The database access routines maintain only one data buffer for each database, which is large enough to accommodate the longest array stored in the data file. Thus, only a small amount of memory is needed for each database. One of the advantages of using byte-stream format files is that they are easy to simulate in memory. We have also experimented with the virtual read and write routines (vread, vwrite) in 4.1BSD UNIX and, thereby, implementing a memory-based database system. Surprisingly, and perhaps unfortunately, the memory-based system was not measurably faster than the disk-based system. The reason for this comparable behavior is that we essentially simulated in software with our disk-based system the same functionality that is provided with the help of the VAX paging hardware on the virtual-memory based system. Since the predominate access time is getting the data from disk, whether this data is paged in with virtual memory assistance or read in via the read system calss makes little difference. Of course, the code to support the former is provided by the system while the code to support the latter was written by us and resides in user space.

The layer of code which resides above the database access routines forms the database editor. We designed a simple command language for it and produced the corresponding grammar. We took advantage of the tools that UNIX offers to generate a prototype parser for the command language. It took about 2 days to produce a parser using yacc and lex. This approach allowed us to test our command language before committing ourselves. The database editor project was a classical software engineering project where we wrote the specifications and documentation, generated code, debugged, and released the product[6].

The user-interaction module is a different story. We did not know at design time all the features that users might want. We therefore attempted to keep the design open-ended. For example, one of the commands in MIDAS is ''run'', which forks off a user-specified shell command, reads back the output, and interprets this output as MIDAS commands. We used this scheme to interface a rigid-body energy minimizer†† with MIDAS; this was done relatively late in the implementation phase but has still proved was to use and powerful. Most users do not realize that the energy minimizer is not a ''built-in'' command to MIDAS. In general, the programming methodology used for the user-interaction module is more akin to the exploratory programming style described in [7] than to classical software development.

The real-time response demanded by MIDAS was provided for in UNIX by some custom kernel modifications written by Tom Ferrin; an rtp() (real-time process) system call was first added to the V6 kernel[8], and later to V7 and 4.2BSD. This system call makes the scheduler always place the calling process at the head of the run queue regardless of other considerations. In addition, time consuming kernel subroutines such as copyseg() and clearseg() are now preemptable. The result is that MIDAS runs fast enough to support real-time interaction. System response for other users degrades significantly while MIDAS is applying transformations to the models; otherwise, MIDAS does not affect system response noticeably.

5. Portability

Tom Ferrin ported MIDAS to VMS with an E&S MPS as the graphics engine. The code in MIDAS is essentially machine-

†† An energy minimizer works by modifying the relative positions of two molecules to decrease the overall energy of the system.

independent C code, which the VMS C compiler handled reasonably well. Ken Arnold ported MIDAS to the SGI IRIS terminal in 4 weeks and to the System V UNIX workstation in an additional 2 weeks. Greg Couch is currently working on a version for the PS300. Our original modular multi-process design has proved extremely valuable in our porting efforts. The database editor process is essentially independent of the different graphics engines and can be brought up on new hardware relatively quickly and without the need of the interactive module. The interactive module, which is very graphics-engine specific, can then be brought up separately and with the knowledge that the database access and editor routines are fully functional. Our experience has shown that if you write programs in a portable fashion, using tools provided in UNIX such as _lint_ and _stdio_, then these programs can be moved to other systems (not necessarily UNIX-based) with relative ease. On the other hand, some programs are necessarily machine dependent. The best approach here is to isolate the machine dependent parts into well known modules and then document the code thoroughly.

6. Future developments

We've reached the point of diminishing returns in terms of adding features to MIDAS. This is not to say that MIDAS is the end of the road; it means that we've taken the minimal resources approach of MIDAS as far as reasonable. However, the availability of more powerful hardware such as VAXen and high performance workstations, new graphics engines like the E&S PS300, and raster graphics workstations such as the Silicon Graphics IRIS, which means that we can provide additional functionality for our chemist and biochemist users. We've already brought MIDAS up on some of our new hardware system, however, it does not take full advantage of the capabilities of these machines. The next generation of molecular modeling system will be much more sophisticated; it will move towards artificial intelligence with some sort of expert system[9] which will free the user from some of the grunt work; and it will take full advantage of the new hardware capabilities.

References

[1] Robert Langridge, Thomas E. Ferrin, Irwin D. Kuntz, and Michael L. Connolly, _Real-Time Color Graphics in Studies of Molecular Interactions_, Science, _211_, 661 (1981).

[2] F.C. Bernstein, T.F. Koetzle, G.J.B. Williams, E.F. Meyer, Jr., M.D. Brice, J.R. Rodgers, O. Kennard, T. Shimanouchi, M. Tasumi, _J. Mol. Biol._, _112_, 535, (1977).

[3] Jeffrey M. Blaney, Eugene C. Jorgensen, Michael L. Connolly, Thomas E. Ferrin, Robert Langridge, Stuart J. Oatley, Jane M. Burridge, and Colin C.F. Blake, Computer Graphics in Drug Design: Molecular Modeling of Thyroid Hormone-Prealbumin Interactions, J. Med. Chem., 25, 785 (1982).

[4] Paul A. Bash, Nagarajan Pattabiraman, Conrad Huang, Thomas E. Ferrin, and Robert Langridge, Van der Waals Surfaces in Molecular Modeling: Implementation with Real-Time Computer Graphics, Science, 222, 1325 (1983).

[5] Conrad C. Huang, Thomas Ferrin, and Laurie E. Gallo, A Tutorial Introduction to the MIDAS Database Tutorial, Internal technical report, Computer Graphics Laboratory, University of California, San Francisco (1984).

[6] Peter Freeman and Anthony I. Wasserman, Software Design Techniques, 2nd edition, IEEE Computer Society (1977).

[7] Beau Shiel, Power Tools for Programmers, Datamation, Feb 1983, pp. 131-144.

[8] Thomas E. Ferrin and Robert Langridge, Interactive Computer Graphics with the UNIX Time-Sharing System, Computer Graphics, 13, 320 (1980).

[9] Robert Langridge and Thomas E. Ferrin, The future of molecular graphics, Journal of Molecular Graphics, 2, 56 (1984).

[10] Laurie Gallo, Conrad Huang, Thomas Ferrin, MIDAS User's Guide, Internal technical report, Computer Graphics Laboratory, University of California, San Francisco (1984).

SPEECH RECOGNITION

Rob Johnston
Zdrav Podolski

University of Glasgow
Computing Science Department
Glasgow
Scotland.

Applications using speech recognition are not widespread, but the large number of published discussions suggest a general interest in the medium. Accounts of the possible use of speech recognition in such advanced application areas as air-force precision bombing [1] imply that we have an advanced and reliable speech recognition technology.

Crusading articles list the supposed gains offered by voice interaction [2], and suggest that the sphere of application of speech input is limited more by the imagination and purse of system designers, than by the remaining technological problems [3].

A closer look at the assumptions surrounding speech input reveals a different picture:

i) Naturalness - this is the idea that speech recognition offers a more natural way of interacting with a computer. The reality is that using a normal tone of voice, particularly with devices that perform whole-word template matching, increases the number of recognition errors. Indeed, it is not uncommon for new users to be instructed to adopt the tone of voice they would use for commands to a dog. Even the more advanced recognisers which attempt to handle continuous speech, nevertheless impose a restrictive syntax combined with a small vocabulary, which is a long way from natural speaking.

ii) Speed - the assumption that speech will offer faster input, simply because people are known to be capable of speaking faster than they can type. In reality, recent experiments [4,5] reveal that replacing keyboard with voice gives significantly slower input, and in particular that speech input more than doubles the time that users spend in checking the screen for errors [5].

iii) Training
- the idea that speech requires no training, because it is already a familiar medium. In fact, the opposite is true. The user has to be trained to speak in the manner required by the system, and most systems have to be laboriously trained for each speaker, and then frequently retrained.

In addition to these general problems there are some particular ones that arise if one attempts to integrate speech into an existing system. As an illustration, imagine that we want to use a spoken subset of UNIX commands:

- some commands are unpronounceable, and may need to be spelt out (pwd, chdir).

- spelt commands may be acoustically too similar (cp, cc, cd, pc).

- allowing single-letter option flags will increase the vocabulary size by up to 26 words, some of which are highly confusable.

- adding new vocabulary words is problematical. Every time a new file name is generated, one would need to to type or spell it the first time it occurred, and the machine would need a short training session on the new word.

In fact, bolting on a speech device to an existing application is always problematical, and may require considerable (and often unacceptable) changes to the command language. In spite of this, and in spite of the imperfections of available technology in comparison with the performance of human listeners, specific areas are emerging where speech input can offer real advantages. Obvious examples are use by the physically disabled and by long-term well-motivated users in 'hands-busy' and 'eyes-busy' occupations; but the growth of interest in human-computer interfacing and dialogue optimisation is likely to expose some other realistic applications using small vocabularies and highly-constrained syntax, where speech may merely augment conventional input modes [5] . Finding out whether an application can use speech to advantage is too often a matter of buying the hardware and hoping it will fit the task.

The search for standards.

What is urgently needed is some way of predicting performance in advance. The only performance figures available until now have been those provided by the manufacturers, normally (and perhaps understandably) measured with a highly-trained speaker in near-ideal conditions, using a small or specially-selected vocabulary. Usually no attempt is made to simulate a real task (because as a general rule, cognitive loading results in degraded recognition performance).

The various standards organisations, such as the NBS in the U.S. have recently devoted an entire conference to the idea of developing objective procedures for speech recognition assessment [6], and they are looking at a number of possibilities:

i) benchmark applications - direct comparison of recognisers using standard vocabularies and syntax.

ii) 'speech databases'
- tape recordings of standard speakers could eliminate the problem of speaker variability, assuming that the organisations concerned can agree on what constitutes a 'standard' accent [7].

iii) user expectations - this approach examines how well a system can meet the requirements of a user. New theoretical models are required - for example the U.S. Navy are developing theoretical models of speech interaction [8], and Logica have suggested a transaction-based model [9].

iv) experimental approaches - this approach is derived from psychology and human factors, and is unique in allowing us to measure those aspects of applications that affect performance. Also, it allows direct comparison with other input modes (usually the keyboard, or perhaps a pointing device and associated software). There are some eighty factors that are thought to affect performance [10], but only a few of these are at all well understood, (e.g. vocabulary size, noise level). Our own work is currently examining two less well documented factors - system training and error recovery procedures.

Implementation.

The hardware to perform recognition is a medium priced, isolated-word recogniser, with a vocabulary of up to 100 words [11]. There is a simple protocol that allows host control of the recogniser, for downloading new vocabularies, setting the recogniser parameters, and so on. These functions are initiated by a series of short C programs, with names like train, recognise, and update.

Two main problems arose at the implementation stage:

The first problem was a human one: the train program builds a new set of templates by prompting the user to utter one word at a time. Subjects using this program typically grew tense and irritable, and were left out of breath after retraining on even quite small vocabularies. It was discovered that this was because the prompt for the next word was being displayed as soon as the previous word had been spoken. This was causing a feeling of urgency in the speaker, and a failure to draw enough breath between words. In short, speakers felt the system was hurrying them. When the program was altered to idle for a second or two between words, the problem disappeared and the resulting vocabularies were also more robust.

The second problem was related to the hardware configuration. The recogniser is designed to share a single serial line with the terminal. While this has the advantage that the mode of input (keyboard or speech) can be transparent to the host, it causes problems in communication of control information between host and device. A user running an application which uses speech input requires:

i) the host should be able to instruct and interrogate the recogniser, without interruption to the application which is also running on the host.

ii) the logic to achieve (i) should not be embedded in the application.

These requirements were fulfilled by using EMACS [12]. Emacs was programmed to handle both input to an application from a user, and input to a recogniser driver program from the recogniser (which is always in packets, with a recognisable header and trailer). Thus, information coming in on a single terminal line is re-routed to one of two EMACS buffers (typically associated with a shell window and an application window).

EMACS was also used to implement a self-updating help buffer, whose contents change in accordance with the state of the dialogue. For instance, the buffer always contains a list of words in the currently active sub-vocabulary. A command-set display like this is even more important for speech than for keyboard input, because of the higher penalties which are potentially incurred by a wild guess made by a user who cannot remember the correct word. The help buffer can be displayed or removed at any time, using the words 'help' and 'no-help'.

Task Area.

The task we are using is correcting and running a Pascal program containing deliberate typing errors. The commands to perform this are provided by a programming environment written by Bill Findlay [13] where students have access to a subset of UNIX commands and some purpose-written commands, via a beginners' shell. The students who will comprise our experimental subjects are already familiar with the keyboard version of this system, so that our results will be only minimally affected by a task learning effect.

A student's environment is constructed so that each programming project takes place in a different directory. Because the naming of files follows the same convention across directories, and because most commands are semantically appropriate to only one or two files, we have been able to do away with command-line arguments. Where some ambiguity is unavoidable, the command becomes interactive, accepting yes-or-no answers. In this way, the vocabulary is reduced from 21 to 15 words, and the average number of words per command from 2.6 to 1.2. These are obviously important gains when using speech input.

Experimental setup.

We now have a UNIX-based experimental testbed for evaluating various hypotheses about the human and systems factors in speech recognition. For instance, a major reason why voice input is so often disappointing in practice is not the fact that it is slow, so much as the tendency to repeat the same error and over, thus bringing the system to an effective standstill. To deal with this problem, a number of error recovery strategies have been suggested, and while there is no difficulty in implementing them, the main work is in evaluating their efficacy. Our current project aims to measure the impact on error frequency and on task completion time.

Conclusion.

Affordable speech technology is not in a state where it can be generally used to replace more conventional input methods. Nevertheless, isolated application areas do exist in which speech input can be used to advantage. In these cases, the user interface has to be carefully designed to take account not only of the application but also of the idiosyncrasies of speech input.

We propose to examine the human and systems factors that apply to speech interaction, in a testbed built on a well-suited application area, a student programming environment.

[1] North, R. A.
'Application of advanced speech technology in manned penetration bombers'
Final report from Honeywell Systems and Research Center, Minneapolis, MN, 1981.

[2] Bridges, A. 'Speech recognition systems emerge for untapped market'
Computer Technology Review, Winter 1983, pp 91-97.

[3] Melnicoff, R.M. (Product Manager, Votan)
'Voice I/O increases user productivity of sophisticated scientific instruments'
Speech Technology, 2, 1, September 1983, 54-59.

[4] Morrison, D.L. et al
'Speech-controlled editing : effects of input modality and of command structure'
Int. J. Man-machine Studies (forthcoming issue).

[5] Damper R.I. et al
'Speech input as an adjunct to keyboard entry in television subtitling'
INTERACT '84 - First conference on human-computer interaction, conference papers Volume 1, pp43-48.

[6] Pallett, D. S. (ed)
Proceedings of the workshop on standardisation for speech technology,
National Bureau of Standards, 19th March 1982, 238 pages.

[7] Vonusia, R.
'NATO AC/243 language database'
in [6], pp 223-228

[8] Harris, S. D. (Naval Air Development Center)
'Voice-controlled avionics: programmes, progress, and prognosis (a case for holistic engineering)'
in [6], pp 111-131.

[9] Peckham, J.
'Automatic Speech Recognition - Matching capabilities with applications'
IERE colloquium on speech I/O, London, 30th March 1984.

[10] Lea W. A.
'Problems in predicting performances of speech recognisers'
in [6], pp 15-24.

[11] Interstate Electronics Corporation SYS-300 operation and maintenance manual (Feb 1983).

[12] 'EMACS - the extensible, customisable, self-documenting display editor'
Proc ACM SIGPLAN/SIGOA Conf Text Manipulation (Portland OR), June 1981, pp 147-156.
UNIX version by J. Gosling of CMU).

[13] Findlay, W., and Watt, D.
'HOCUS - a UNIX subsystem for novice programmers'
University of Glasgow, Computing Science Department, Technical Report CS/84/R4.

Processes as Files

T. J. Killian

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We describe a new file system, **/proc**, each member of which, **/proc/**nnnnn, corresponds to the address space of the running process whose pid is nnnnn. Access to these files is restricted, via the normal file protection mechanism, to the process owner. *Lseek*(2), *read*(2), and *write*(2), allow inspection and modification of the process' image. Other services are available via *ioctl*(2), including stop/go on demand, selective intercepting of signals, and the ability to obtain an open file descriptor for the process' text file. The technical problems related to the implementation of **/proc** on a VAX† under the 8th Edition of the Unix‡ operating system have mostly to do with the paging system. Security issues are also considered.

The window-based interactive debugger *pi*, developed by T. A. Cargill, is the first major user of **/proc**. It can control multiple processes dynamically and asynchronously. Thanks to the network file system, **/n**, these processes may be running on several different machines. We also describe an efficient, almost portable ps(1).

Introduction

Any debugger is dependent on, and often limited by, its ability to access the address space of the debugged program. This is especially true in the case of interactive debugging under the Unix system, where the debugger and the debugged object are separate processes. The problems associated with the standard mechanism, *ptrace*(2), are well known:

† VAX is a trademark of Digital Equipment Corporation.
‡ Unix is a trademark of AT&T Bell Laboratories.

March 17, 1985

- The object must agree explicitly to be debugged, and furthermore it can only be debugged by its immediate parent. Thus there is no dynamic binding, and children of the original object process cannot be handled.

- Before it can be examined, the object must be put in a stopped state, typically by sending it a signal(2). This can interrupt the object's own system calls, so the debugging is not transparent. Or the object may be ignoring signals (e.g., sleeping forever on a locked inode), so the mechanism can fail entirely.

- Ptrace(2) provides low bandwidth at high cost: two context switches per word of data transferred, an achievement equaled only by some text editors. Its protocol is arcane and unnatural compared to most other system calls.

We have tried to overcome these difficulties by providing an interface that is as uniform as possible, using an existing mechanism for accessing random data external to a process: the file system.

Fig. 1: A sample /proc directory

```
-rw-------  1  root        14336 Feb 20 12:59 00001
-rw-------  1  root       528384 Feb 20 12:59 00002
-rw-------  1  root        12288 Feb 20 12:59 00019
.
.
-rw-------  1  tom         32768 Feb 20 12:59 02596
-rw-------  1  tac        106496 Feb 20 12:59 02652
-rw-------  1  root        14336 Feb 20 12:59 02801
-rw-------  1  tac         39936 Feb 20 12:59 02900
-rw-------  1  tac         23552 Feb 20 12:59 02910
-rw-------  1  tac        184320 Feb 20 12:59 02911
-rw-------  1  tom         33792 Feb 20 12:59 02912
-rw-------  1  tom         54272 Feb 20 12:59 02913
```

System-Call Interface

Fig. 1 shows the result of a typical ''ls -l /proc.'' The name of each entry in the directory is a five-digit decimal number corresponding to the process id. The owner of the ''file'' is the same as the process' user-id; note that only the owner is granted permissions. The size is the total virtual memory size of the process. The time is not very useful: it is always the current time, for reasons discussed later.

The standard system-call interface is used to access /proc. Open(2) and close(2) behave as usual with no side-effects. In particular, the object process is not aware that it has been opened. Data may be transferred from or to any locations in the object's address space through *lseek*(2), *read*(2), and *write*(2). Reading and writing have slightly peculiar behaviour due to the segmenting of the process' address space.

Fig. 2: Address structure of /proc/*nnnnn*

```
           |                        |
0x80000000|     System segment     |
  --------+------------------------+
           |                        |
0x7ffff000|       User area        |
  --------+------------------------+
           |                        |
      (*)  |     Stack segment      |
  --------+------------------------+
           |                        |
           |     [Non-existent]     |
      (*)  |                        |
  --------+------------------------+
           |                        |
      (*)  |      Data segment      |
  --------+------------------------+
           |                        |
0x00000000|      Text segment      |
  --------+------------------------+
```

(*) addresses computed from segment sizes

The *text*, *data* and *stack* segments (see Fig. 2) all allow both read and write access; if the text segment was shared at the time of the write, a private copy will be made. The *user area* is read-only, except for locations corresponding to saved user registers. The system segment is not accessible. For simplicity in enforcing these restrictions, a single I/O operation may not cross a segment boundary; the byte count will be truncated if necessary. Note that for ordinary files, the entire file is either write-protected or not, and ''holes'' read as zeroes.

As with other special files, there are a number of services available via *ioctl*(2), in this case having to do with process control:

PIOCGETPR	fetches the object's struct proc from the kernel process table. Since this information resides in system space, it is not accessible via a normal read.
PIOCSTOP	sends the signal SIGSTOP to the object, and waits for it to enter the stopped state.

PIOCWSTOP	simply waits for the object to stop.
PIOCRUN	makes the object runnable again after a stop.
PIOCSMASK	specifies (via a bit mask) a set of signals to be *traced*; i.e., the arrival of such a signal will cause the object to stop. A mask of zeroes turns off the trace. There are three side-effects: (1) a traced process will always stop after *exec*'ing; (2) the traced state is retained after the object is closed, although the mask bits themselves are lost; (3) the traced state and mask bits are inherited by the child of a *fork*(2).
PIOCCSIG	clears the object's currently pending signal (if any).
PIOCOPENT	provides, in the return value of the ioctl, a read-only file descriptor for the object process' text file. This allows a debugger to find the symbol table without having to know any path names.

All system calls are interruptible by signals, so that, for example, an *alarm*(2) may be set to avoid waiting forever for a process that may never stop. Any system call is guaranteed to be atomic with respect to the object, but, as with ordinary files, there is nothing to prevent more than one process from trying to control the same object.

Implementation

The interface to the file system was provided by P. J. Weinberger, who introduced the notion of a *file system type* to support his network file system [1]. In a sense, this is a generalization of the top level of the *pipe*(2) mechanism, in which, e.g., the kernel's read routine calls a special procedure if the file descriptor refers to a pipe. In Weinberger's scheme, the kernel has a well-defined set of internal entry points (*read*, *write*, *open*, *close*, etc.) for each file system type, and uses the appropriate one transparently. A special *mount*(2) command is used to associate a particular file system type with a given leaf of the directory hierarchy. Weinberger in fact made an early attempt at an implementation of **/proc**, but the communication mechanism was too similar to that used by *ptrace*, making it impractical.

In our implementation, the calling process accesses the object directly. This requires the cooperation of the swapper, the scheduler, and the paging system. A flag in the object's struct proc informs the system globally that the object is undergoing I/O via **/proc**. The swapper recognizes the object as a candidate for being swapped in, and will not swap it out as long as the flag is set. The scheduler will not run the object; in particular, the effect of any signal or wakeup on the object will be delayed until I/O is completed. Finally, hooks have been added to the paging system so that the object's pages may be brought in on demand by the calling process. Thus I/O via **/proc** can take place under almost any circumstances. The exceptions occur where the object is waiting for an actual virtual-memory event, i.e., it is exec'ing, paging, forking, or exiting. In these cases, the I/O call returns an error, and the caller must try again.

Because the caller and object must both be swapped in during I/O, there is the possibility of deadlock on very small systems. In practice, this is unlikely to happen under 8th Edition Unix because being swapped in implies only that the user area and page tables are present; the remainder of the process can page in and out as necessary. In any case, I/O via **/proc** is always interruptible, even in an otherwise deadlocked situation.

Reading the **/proc** directory and _stat_'ing(2) its members present special cases. The information returned is made up exclusively from the kernel's process table. Some potentially useful data, like process times which reside in the user area, are not returned for efficiency reasons, since a swap might be required to get access.

Security

The most obvious security loopholes are plugged by the file system interface. The standard protection mechanism prohibits free-for-all access to every process, and the file system type itself does not support things like _mv_(1), _rm_(1), _chmod_(1), etc. Some more subtle problems are the following: **/proc** provides a way of reading a file which has execute, but not read, permission; this is easily solved, since the inode of the object process' text file is available at the time the object is stat'ed or opened, and permissions can be checked. If a process opened by **/proc** exec's a program with setuid bits, there is the potential for any user to become the super-user; we take care of this by not honouring the setuid bits in such a case. Note that if the open is attempted after the exec, the process owner will have already changed and the normal protection applies.

Finally there is the problem of not corrupting shared text. The **/proc** interface does copy-on-write, as mentioned previously. Due to the page-table scheme used by the VAX, it is simplest to copy the new private text segment directly from the text file. A special check is necessary at the time of a _fork_. Breakpoints already set in the parent should be inherited by the child, and yet subsequent modification of the parent should not affect the child; thus a copy from the parent's address space must be made in this case. There is one more case, added as a convenience: if an opened process exec's, it is automatically given impure text, on the assumption that a debugger is waiting in the wings.

The system interface

T. A. Cargill has developed an interactive, window-based debugger called pi (process inspector). It can be bound dynamically to multiple processes, and control them asynchronously. Binding is particularly simple: pi opens the object process, gets hold of the text file via PIOCOPENT, and reads the symbol table. It is frequently useful to have control of the object before it has actually begun executing. This is easily done, given the semantics of PIOCSMASK. Fig. 3 shows the program hang, which takes any command as argument, and starts it up in the stopped state.

Fig. 3: The hang program

```
£include <stdio.h>
£include <signal.h>
£include <sys/pioctl.h>

main(argc, argv)
int argc; char **argv;
{
    int pfd; char procnam[16]; long mask = (1<<(SIGSTOP-1));
    FILE *ttyerr;
    ttyerr = fopen("/dev/tty", "w");
    if (argc <= 1) {
        fprintf(ttyerr, "Usage: %s cmd [args...]\n", *argv);
        exit(1);
    }
    sprintf(procnam, "/proc/%05d", getpid());
    if ((pfd = open(procnam,0)) < 0) {
        fprintf(ttyerr, "cannot open %s\n", procnam);
        exit(1);
    }
    ioctl(pfd, PIOCSMASK, &mask);
    close(pfd);
    fprintf(ttyerr, "%s\n", procnam);
    fclose(ttyerr);
    execvp(argv[1], argv+1);
    perror(argv[1]);
    exit(1);
}
```

Pi obtains status information and data from a process using ioctl's, or a combination of seeks and reads. Other operations require a more complicated series of primitives. Single-stepping is accomplished as follows (the object is assumed to be already stopped):

1) Arrange, via PIOCSMASK, for the object to stop on SIGTRAP.

2) Set the TRACE bit in the object's PSL. This involves a seek, read, and write in the object's user area.

3) Issue PIOCRUN.

4) Wait for the object to return to the stop state, by issuing PIOCWSTOP.

5) Issue PIOCCSIG to clear the SIGTRAP.

Single-stepping over a function call (CALLS instruction) is only slightly harder. If the instruction stepped was a call, one sets the TRACE bit in the PSL saved on the stack and issues PIOCRUN; the SIGTRAP will then be taken upon return.

Breakpointing is similar to single-stepping. In step (2) above, one writes a BPT instruction at the desired location.

The command

We have written a version of *ps*(1) which is two to four times faster than the standard one. It uses the **/proc** interface to obtain process table information, obviating the need for searching the kernel symbol table. Usually the struct proc contains sufficient information to decide whether or not the process warrants further inspection. If so, the SLOAD bit indicates whether it is more efficient to read the user area and stack segments from **/proc**, or from **/dev/drum**. Fig. 4 sketches the main flow of control for *nps*.

Summary

We have described a uniform mechanism for transparent, dynamic communication between a debugger and its debuggee, along with a set of primitives for process control. It is successful largely because it fits so well into the Unix model. We feel that it is probably ill-suited for general inter-process communication, though it should work well for specialized application-oriented debuggers, or other programs which would benefit from clean access to dirty data structures.

Acknowledgements

It is a great pleasure to acknowledge many useful and stimulating discussions with T. A. Cargill, D. M. Ritchie, and P. J. Weinberger.

Fig. 4: Skeleton of the nps program

```
£include <sys/param.h>
    . . . . . . . .
int drum;

main()
{
    int dirfd; struct dir dir;
    drum = open("/dev/drum", 0);
    chdir("/proc"); dirfd = open(".", 0);
    while (read(dirfd, &dir, sizeof dir) == sizeof dir)
        if (dir.d_ino && dir.d_name[0] != '.')
            do_proc(dir.d_name);
}

do_proc(s)
char *s;
{
    int fd; struct user u; struct proc proc;
    fd = open(s, 0);
    ioctl(fd, PIOCGETPR, &proc);
    if (proc.p_flag&SLOAD) {
        lseek(fd, 0x80000000-UPAGES*NBPG, 0);
        read(fd, (char *)&u, sizeof u);
    } else {
        lseek(drum, proc.p_swaddr*NBPG, 0);
        read(drum, (char *)&u, sizeof u);
    }
    print(fd, &proc, &u);
    close(fd);
}
```

References

[1] Weinberger, P. J. ''The Version Eight File System,'' in Proceedings of the Summer 1984 USENIX Conference, Salt Lake City, Utah.

UNIX in Australia, 1984

J. Lions

University of New South Wales

1. The Australian University Scene

In three months the University of New South Wales will be celebrating the tenth anniversary of its receipt of the Fifth Edition of UNIX. Ten years ago, UNIX was not yet, as it is now, known as a trademark of AT&T Bell Laboratories. The exact chronology is now hard to reconstruct - no one then understood how far this new undertaking would carry us - but our license agreement was dated December 15, 1974, and we received our tape and manuals by air-freight shortly thereafter, just in time for Christmas.

Within the Computer Science department of the University of New South Wales, we found that UNIX provided all sorts of new insights into traditional problems and into some new ones as well, such as word-processing, that we hadn't previously thought much about. My own first serious project on UNIX was for the production of a newsletter named ''CUSwords''. This was an appropriate name for a newssheet for a Computer Users Society on campus whose main aim was to pressure the university's central computing centre into providing more useful services occasionally. It was the continuing insistence of the computer centre to feed everyone on a raw diet of Fortran and not much else that made us determined to get our own full-blown UNIX system to be used for all aspects of Computer science teaching. Australian universities have differed from UK universities in that funding for computing has always been a local decision and there has never been a centrally funded, coordinated national plan for the provision of computing services.

Many other departments of Computer Science around Australia have since followed our lead in basing their major Computer Science activity on UNIX. Our first outside 'convert' was Piers Lauder of Sydney University. Another early one was Juris Reinfelds of Wollongong University (just after he had signed up for Interdata kit, because DEC was too expensive).

The battle at the University of New South Wales for provision of adequate facilities peaked in 1979, when students often could not find a free terminal at midnight, and sometimes only with difficulty, at 2 am. By comparison, Sydney University's problem peaked only last year when they sometimes had more than 100 users logged into a single VAX computer. Now they have a second VAX, for staff use, and the student load on the original VAX usually peaks at around 80 users.

It is for reasons such as these that Australian UNIX users have long been concerned with maximum operational efficiency more than increased functionality.

1.1 Highly Tuned Systems

Right from the start, we found ourselves using systems that were chronically overloaded, and by users with real deadlines (student assignments are like that). We had neither the luxury of shedding load (sending our students back to the central computer centre was not an option) nor the luxury of buying additional hardware if and when required.

Changes made at the University of NSW in the 1978-79 period by Ian Johnstone, Greg Rose and others converted a situation where the CPU was spending 40% of its time in user mode, and 60% in kernel mode (leaving 0% idle), to one where the user/kernel split was reversed (60:40). The effective 50% improvement certainly helped. Ian Johnstone subsequently moved to Bell Laboratories, and some of the changes (hashed inode tables for example) that we made then have finally made it into System Five.

There is still no shortage of ideas for further tuning and improvements. Recently Robert Elz of Melbourne University has been experimenting with improving directory searches. Some heavily used programs ('ls' in particular) follow what is in effect a quadratic algorithm when searching directories (they read the directory, and then perform on each file an operation such as 'stat' that implies a directory search). Kirk McKusick at UCB has changed the kernel to keep an 'end-of-last-directory-search' offset value, one per process, and to initiate new directory searches by resuming from this offset value (of course looping back to the beginning of the directory if required). Robert has found that implementing a cache for recently found names in the directory structure can also expedite searches in up to 75% of cases. These two ideas seem to mesh very nicely, as the cache benefits the search for the early components of a path name, and the directory offset benefits the search for the last component.

1.2 Improving Disk Accesses

It is well known that disk accessing can be a limiting factor in system performance, and this has been tackled by UC Berkeley primarily by increasing the basic unit of disk allocation by a factor of two or more. By reducing the number of accesses to disk this speeds up the average disk transfer rate, but at the significant cost of reducing the efficiency of disk storage utilisation by some 15-20% as file sizes are rounded up to multiples of 1024 or more.

Both Sydney University and ourselves have felt that we could not afford to buy the Berkeley changes in total, and this one in particular. Tim Long, then with Sydney University, found recently that disk throughput could be substantially improved without changing the long-established disk allocation quantum (512 bytes) and file system structures. His changes are as follows:

1. keep the segment of the free list (of disk storage blocks) stored in the superblock for a file system in ascending order. This will tend to localise in space the blocks allocated to any file that is written locally in time (i.e. during a short time period).

2. when a file is apparently being read sequentially, the number of read-ahead blocks i.e. blocks read into the buffer pool in anticipation of future use, may be varied from one to thirty-two. Any block read is placed at the head of the 'bfreelist' and will gradually work its way to the end of the list whence it may be reused. The policy that Tim chose was as follows: when the time comes to interrogate the contents of a buffer containing a read-ahead block, determine how far it has progressed in the freelist. If it has progressed by less than one eighth of the list, double the number of read-ahead blocks for that file; if it has progressed by more than one quarter of the list, halve the number of read-ahead blocks.

3. in the disk device driver, when beginning access to a new cylinder, extract from the device queue all the outstanding commands queued for that cylinder. Sense the position of the disk and pick the operation that can be performed soonest. As far as possible, combine two or more operations together. On machines such as the VAX with scatter read/write capabilities, this can work particularly well when several blocks are to be read from the same track. The contents of any disk block that is not required but lies between the first and last required blocks may be read into one of the yet unfilled buffers in main memory, and then immediately overwritten. It is Tim's impression that it is worthwhile skipping single individual blocks on a track in this way, but that where two or more unneeded ones occur together, it is preferable for the driver to read the first block in one operation, and then to be reinvoked to reconsider its next operation (which frequently, but not always, will be to read the

next block on the same track).

These changes have been found to double disk throughput on average. In extreme cases speed improvements of up to a factor of five have been observed while reading some sequential files on an idle machine.

1.3 **Share Scheduling**

This is an appropriate occasion to acknowledge our debt to the University of Cambridge for the Share Scheduling scheme. A colleague, David Hunt, spent a sabbatical leave here in 1975 and brought back the news of the effective scheduling scheme developed by J. Larmouth for use in a batch processing environment.

Although the details do not carry over immediately to a time-sharing system, the basic philosophy does: a user should be able to consume the system's resources at a rate proportional to his current 'share'. The latter can be determined based on the user's intrinsic worth (as determined by the management) discounted by a factor related to the resources he has consumed in the recent past.

Reevaluation of shares and rates of working are needed at moderately frequent intervals. The system may adjust a component of the user's 'nice' value to align his rate of working with his computed 'share'. Individual user shares can fluctuate markedly as other users login and logout. Users tend to have a high share when they first login, but this can decline rapidly once they start to perform useful work. This scheme has been successful in making a single machine appear lightly loaded to a deserving user, while at the same time appearing very sluggish to an undeserving one.

The original implementation of share scheduling to create the Australian Unix system Share Accounting Method (AUSAM) was done by Andrew Hume before he joined Bell Laboratories. It was later adopted by Sydney University and further developed there. There has been some disagreement about some implementation details: currently the University of NSW does not use the share scheduler, the University of Sydney swears by it (if not at it) and it has recently been modified and installed by Robert Elz of Melbourne University into their 4.2+ BSD system under the acronym of 'MUSH'.

1.4 Epicondolitis

No discussion of our current scene would be complete without mentioning an isolated but severe epidemic at Sydney University of epicondolitis (sometimes known as tenosynivitis, or 'repetition strain injury'). No less than three academics have had to take sick leave, one for two months. There are at least two common factors: they have offices on the same corridor, and they use the same UNIX system.

2. My Activities

I would like to dwell for a moment on my own activities. While these are just a few of many interesting UNIX applications in and around Australia, they have the advantage of being familiar.

As editor of a scientific journal, I am much concerned with ways to expedite an extensive correspondence, much of it very routine. My efforts to devise effective procedures recently culminated in the first version of a software package that I call 'correspond'. It is composed for the most part of a careful formulation of shell procedures that use standard commands. My conclusion is that, if one discounts 'nroff' for a moment, the standard UNIX tool-kit is capable of solving all the relevant problems in a satisfactory manner.

My other application is an annual one: for me, teaching operating systems means also teaching C. A new language is best learnt via at least one programming exercise - in this case a suitable exercise is for a program that interacts with its environment via system calls, especially 'fork' and 'exec'. My annual challenge comes from the need to assess the resulting programs. Each year I try to become more cunning in setting the ground rules for the exercise, but each year some new mutation of Murphy's Law appears. My shell procedures have grown very elaborate; I use every trick I know with 'awk', 'sed', 'grep', 'diff', etc. to try an discipline the students' output for my test data. I have concluded that, for this purpose, the standard UNIX tool-kit is far from complete: in particular I need a file comparison routine that is far more sophisticated than 'diff' i.e. a real expert system.

2.1 Drafting the AUUG constitution.

Recently I have been involved in drafting a constitution for the Australian UNIX systems User Group. This was an illuminating experience. I wonder how many of you have noticed the similarities between such documents and computer programs. A constitution is a program that is executed, with variations, by many independent, sometimes cooperating, sequential processors. The rules for exception handling are particularly interesting: there are more defaults, with much more interesting precedents, than any PL/1 programmer ever dreamed of. I think both programmers and lawyers could gain by exploring the common ground that can be found here.

2.2 AUUG

On August 27 the AUUG became a formally constituted society, with a management committee of seven members and myself as President. Its initial activities will be as before: to publish a newsletter (six issues per year) and to hold twice-yearly technical meetings. We shall be delighted to welcome you all as subscribers to the newsletter and as visitors to our meetings.

3. ACSNET

In a country as full of train-spotters as England, it should not be necessary to explain to many of you that, due to lack of foresight by our pioneering forefathers, Australia still has railways built to three separate gauges, with all the inconvenience and expense that that entails. However we are hoping that history won't always repeat itself.

The analogue today of last century's battle of the gauges is the 'battle of the protocols'. One jousting ground is the world-wide community of UNIX users who wish to communicate with each other. Our champion knight in these lists is known variously as SUN or ACSnet. SUN (for Sydney UNIX Network) is the technology; ACSnet (for Australian Computer Science network) is one implementation of this, much in the same way as USENET is one exploitation of UUCP.

Uucp has been one of Mike Lesk's greatest ideas. Initially it allowed a UNIX machine with an automatic calling unit (ACU) to communicate as required using the telephone network with many other UNIX machines within Bell Laboratories. At first, all communications were direct, and involved only the source and destination machines. However, even within Bell Labs, not every UNIX machine is blessed with its own ACU, and occasions arose when two such machines were required to communicate. This could be arranged if a third, ACU-equipped machine participated in a two-stage transfer. From there, the jump to multi-stage transfers along with multipart destination names was easy.

One well-known problem with networks based on uucp is that messages must be explicitly routed. This would not be so bad except network topology can, and will, change almost daily, and nodes may fail, or become congested or just plain uncooperative.

In Australia, when we built our first UNIX network, starting about 1979, things were entirely different from the Bell Labs situation: ACUs were non-existent, connections were via tie-lines, and each host was connected only to one or at most a few other hosts. Multistage transfers were normal, but multipart names were not. Only the final destination had to be named in the message, with the network providing the necessary routing information. Moreover single physical channels were multiplexed into sets of virtual channels to enable remote logins as well as file transfers over a single tie line.

Recent developments to SUN version III include changes to improve robustness, make proper use of full-duplex connections and ACUs, give better support for mixed message flows, and a more general scheme for naming hosts using the concept of 'domains', which becomes almost mandatory once the number of hosts grows beyond the ken of any single host. The name/address couple 'name.host' may be generalised to 'name.host.domain' (or even more generally to 'name.host.domain1.domain2...domainN') when the host name is not unique, or routing information for the domain, but not for the particular host is known at the source machine. The form 'name.host' may also be generalised to the form 'name.domain' if a name server exists for the domain.

Development of the SUN software is being undertaken by Piers Lauder and Bob Kummerfeld of Sydney University. Development of an X.25 interface that will provide virtual circuits for use by ACSnet is being carried out by Robert Elz at Melbourne University.

Recently Michael Rourke at the University of NSW has written a program 'netpath' that will attempt to locate an optimal path between any pair of hosts known in a database of UNIX-machines worldwide that has been set up by Peter Ivanov. I am now agitating for a program that will provide complete ready-to-use mail addresses for my overseas correspondents that are separated from me not by oceans but by wildly diverse computer networks.

4. Non-DEC Hardware

One of the interesting new developments here is a growing flight away from DEC hardware. New suppliers in the market place with processors faster than the VAX 11/780 have opened up interesting new avenues for consideration, and the incentive to explore these has been helped along recently by DEC's maintenance support.

4.1 The Monash Pyramids

Monash University in Melbourne bought two early models of the Pyramid: they got a good price, and though they did not plan to be, they have effectively functioned as a Beta-test site. Since there is likely to be some interest in these machines, I thought a brief summary of results reported by Ken McDonell of Monash University at the recent AUUG meeting might be of interest:

''The Pyramid architecture features a blend of novel RISC-based ideas, bit-slice componentry and third-party input-output subsystems. ... Pyramid's UNIX port (OSx) is innovative in that it attempts to provide concurrent support for both AT & T's System V and Berkeley's 4.2BSD versions. This ... relies upon an extension of the Berkeley symbolic link concept ...''

After some drawn-out, initial teething problems, Monash is pleased with its Pyramids. There are still problems inherent in the absence of VAX-dependent software, missing utilities from the System V set, and limitations in connecting peripherals to the i/o bus, which is not a true Multibus as advertised. Much tuning and refinement remains to be carried out, but will almost certainly improve the present advantage of the Pyramid, which can be summarised as ''1.35 times the power of a VAX 11/780 for two-thirds the price''.

4.2 Monash Bench-mark Suite

As part of their initial selection procedures, and part of their ongoing effort to keep their supplier honest, Monash University has developed an elaborate Benchmark Suite known as 'MUSBUS'. As benchmarks are really only interesting to the cognoscenti, it is not appropriate to dwell on this one here, except to say that it exists, it attempts to assess arithmetic capabilities, recursion, context switching, interesting system calls including 'fork' and 'exec', memory accessing, filesystem throughput and multi-user interference effects. More information, if you are interested, can be obtained from Ken McDonell.

4.3 ELXSI Port

While the porting of UNIX to yet another computer system is no longer particularly newsworthy, one such activity currently being carried out in Sydney certainly interests me, since it is being carried out by a small group that is headed by one of my former students, Greg Rose, and that has succeeded where another group (nameless) had failed. Here the target machine is the ELXSI 6400, born in Silicon Valley to a new breed of computer architects who think big:

1. It is a tightly coupled multiprocessor system, with each processor rated at about 4 Mips (i.e. about four VAX 11/780s). A fully expanded system contains ten CPUs.

2. The system interconnection is via 128 bit wide bus with a raw speed of 320 megabytes per second, and usable speeds up in the 200 mbps range.

3. The basic architectural style is to use messages for all communications between processes, processors and hardware peripheral controllers. The speed of the common bus is sufficiently high that contention for its use is rare, so that interprocessor interference is rare, and system power grows almost linearly as processors are added (at least that is the claim: Greg tells me that he believes it).

The ELXSI port is still at an early stage: the current C compiler, without the second optimising pass, generates code that shows the ELXSI to be 2-3 times the power of the VAX (the optimising pass should increase its speed by at least 20%). Because UNIX is being built on top of the preexisting ELXSI operating system, EMBOS, certain things are less than perfect, particularly the 'fork' system call, which is still too costly in time. Also, incremental dumps have proved a problem since an entire UNIX file system is equated to a single EMBOS file.

There are some advantages: two or more UNIX systems can exist simultaneously within the one set of hardware (which is certainly attractive to developers). Also the system library is truly shareable, so that the code size for many of the standard UNIX commands drops significantly.

There have been some difficulties with integers: the hardware supports 64 bits nominally, but handles 32 bit integers better in some situations. As things stand, 'short' translates to 16 bits, 'long' to 64 bits, and 'int' to 32 bits. I am told there are places in the system where it is assumed that while an 'int' may be either 'short' or 'long', it is expected to be one or the other.

The ELXSI port is interesting, it does address the high end of the market. It is comparable with, but less expensive than, the Cray 1 that Bell Labs Computing Research is buying. Melbourne University has now installed one and I think you may hear more about the ELXSI 6400 in the future.

4.4 The First UNIX Port

I said earlier ''the porting of UNIX to yet another computer system is no longer particularly newsworthy'' but there had to be a first time. It may not be widely known, but in the period 1976-77 there were two efforts in progress to port UNIX from the PDP11 to another machine. By coincidence the target in each case was an Interdata machine. One of these efforts is well-known, involved members of Bell Laboratories, led to the development of Level Seven UNIX, and was well reported in the 1978 special issue of the Bell System Technical Journal.

The other effort was carried out at Wollongong University, against great odds, by one person working almost without assistance and few resources. Although Richard Miller has now returned to his native Canada, we are proud to say that this work was done in Australia and that we - particularly Juris Reinfelds - can take some of the credit (or blame if you prefer) for creating the conditions under which he worked.

You may be interested to know that Richard has been nominated for ACM's prestigious Grace Hopper award. The result of the nomination will not be finally known until next November, but if he is successful, it will be the first time that the award has gone to someone other than an American. I hope you will join me in keeping your fingers crossed.

5. UNIX goes commercial

So far I have not said much about the commercial exploitation of UNIX within Australia. It is happening but it doesn't seem to have occurred at the same rate as elsewhere. The Australian commercial community has tended to stick to its guns and sales of traditional name-brand mainframes have remained as strong as ever.

However this should change rapidly, since our universities have for some time (misguidedly in the opinion of many) been turning out graduates well-acquainted with UNIX and almost totally unacquainted with the likes of systems such as MVS. There has always been a straightforward justification for this: we could afford UNIX but not MVS. We have been surprised that others did not reach the same conclusion soon after we did.

There is now strong recruiting in Australia of graduates with UNIX and C experience for overseas positions. Moreover, since many of the students now crowding into our Computer Science courses come from the near North (or if you prefer, the far East), if you are interested in looking to recruit such graduates you might do worse than look towards Malaysia, for example.

5.1 **Commercial Activities**

As examples of current commercial activities, I might mention just two items:

1. Time Office Computers in Sydney have embarked on an ambitious project to build a distributed system consisting of high performance workstations (up to several hundred) communicating via Ethernet and running UNIX as their basic software.

2. Qantas Airways has acquired a Pyramid 90x to develop and support the sale of tour packages. Eventually they hope to serve up to 80 operators world-wide. The application uses the 'Unify' relational database package and after six man-months and about 15,000 lines of C code they have completed about 80% of the project. (This may be compared with the estimated 30 man-years expected for their more traditional mainframe approach.)

5.2 **A UNIX Software factory**

While commercial exploitation of UNIX as a base for computer applications may have been slow to start, activities in UNIX systems development have been moving ahead. Greg Rose (already mentioned in connection with the ELXSI port) has been one of the principals of a small company called Fawnray, that has already completed several successful ports for IDRIS and UNIX.

His company has recently suddenly become much larger by the infusion of substantial Government-backed venture capital. The goal is to provide strong UNIX software systems backup for many fledgling hardware companies forming in Australia. Although you may groan ''not another M68000 port!'', we believe that it is important that proprietary rights to UNIX software enhancements should be owned locally so that Australian hardware companies will not be beholden to and held to ransom by remote overseas companies, for whom Australia is right off the edge of the map.

6. In conclusion ...

I feel bound to say that, in 1984, in Australia, the UNIX system is alive, well and quietly humming along.

Acknowledgements

The assistance of Robert Elz, Peter Ivanov, Piers Lauder, Tim Long, Ken McDonell and Greg Rose with the preparation of this paper is gratefully acknowledged.

Connecting UNIX* Systems Using a Token Ring

Robbert van Renesse
Andrew S. Tanenbaum

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam

Sape J. Mullender

Centre for Mathematics & Computer Science
Amsterdam

ABSTRACT

As part of the research on distributed operating systems being done at the Vrije Universiteit, we have implemented a set of network-oriented programs for use on several UNIX machines connected by a high-speed token ring. With these tools it is possible to transfer files between machines, log in to remote machines, and implement multimachine shell scripts. The transaction protocols discussed in another paper at this EUUG meeting are used to implement two basic services: a "shell server" and a data transfer service. Other services are easily implemented as shell scripts that use these services. A file transfer program, for instance, executes the command "to < file1" on one machine, and "from > file2" on the other machine. More examples of these facilities and their implementation and performance are discussed in the paper.

1. Introduction

At our university we are developing a distributed operating system called **Amoeba**[Tanenbaum81] As a spin off from this research, we have incorporated some of the **Amoeba** interfaces into UNIX, and used these interfaces to build some application programs for communicating between UNIX systems. These tools include file transfer, remote execution and remote

*UNIX is a Trademark of Bell Laboratories.

login. In this paper we describe the different layers into which our implementation is divided, and the interfaces that connect them, and discuss the performance of our implementation.

When we started this project we had 2 PDP11/44's running UNIX V7, 2 VAX 750's running Berkeley 4.1BSD and 8 Intel 8086's and 8 Motorola 68000's running **Amoeba** 1.0. As **Amoeba** was designed to be a distributed system, we needed a network.

Our network had to be fast, even under heavy load, so a ring network seemed the best choice. After some study, we chose ProNET. This is a 10 Mbit/sec star shaped ring network with decentralized control and token arbitration, supporting up to 255 hosts. It can send and receive packets concurrently, do scatter/gather operations, has variable length packets up to 2044 bytes, checks parity, and has a primitive hardware acknowledgement bit. Pronet interfaces exist for UNIBUS and MULTIBUS; both are used in our machines.

Our desires, with respect to UNIX, were modest. We did not want to make a distributed system, but only some capabilities to do file transfer and remote execution. In retrospect, we feel that we have achieved these objectives.

2. Network Interface

Network application programs need a mechanism to commmunicate reliably. We have designed a network interface that is simple to use, which uses a efficient, simple and fast protocol. We envision communication between two processes, one is called the server and the other the client. A server handles requests from clients. When the server has handled the request it sends a reply back to the client; the sending of a request to the server and a reply back to the client is called a transaction*[Mullender84]

The transaction primitives are:

```
        typedef struct Mref {
                char     *M_oob;
                char     *M_buf;
                unsigned  M_len;
        } Mref;
```

PDP, VAX and UNIBUS are registered trademarks of Digital Equipment Corporation.
ProNET is a trademark of Proteon Associates, Inc.
MULTIBUS is a trademark of Intel, Inc.
*Not to be confused with the concept "atomic transaction."

The client, in order to do a transaction calls

```
trans(cap, req, rep);
        Cap *cap; Mref *req, *rep;
```

The server receives requests and sends replies with

```
getreq(port, cap, req);
        Port *port; Cap *cap; Mref *req;

putrep(rep);
        Mref *rep;
```

3. User Programs

And now the moment of truth: can the primitives we designed be used to make useful programs? The basic things we want are file transfer and remote execution. In this section we will discuss some of the programs we have built; they fulfilled our desires and are now among the most-used programs on our UNIX systems.

3.1. File Transfer

The first thing expected of a fast local network is fast file transfer. We have made two simple programs to accomplish basic data transfer, requiring the user to be logged in on both the machine producing the data, and the machine consuming the data. Their syntax is:

```
from identifier
to identifier
```

To reads from standard input and *from* writes to standard output. If the identifiers of *to* and *from* are the same, the input data to *to* becomes the output data of *from*.

For example, when "to hamlet < /etc/passwd" is executed on machine A, and "from hamlet > /etc/passwd" on B, the password file of machine A is copied to the password file of B. The same can be done with the execution of "rcp A!/etc/passwd B!/etc/passwd," called at any machine on the network; *rcp* will be treated in a later section.

3.2. Remote Login

As programmers are lazy, they do not like to walk from terminal to terminal to work on different machines, especially if the terminals are in different rooms, floors or buildings. So a desire existed to be able to login onto any computer from any terminal; therefore, we made our own version of the *cu* command to call another UNIX system, except that our version does not lose characters. The syntax is:

```
        call machine-name
```

After calling this program you get a login message from the remote machine, and you can login and work onto that machine as if the terminal is connected directly to the new machine, with one exception: lines beginning with a '~' are special. Their meaning is as follows:

```
        ~.: switch back to local machine;
        ~!: shell escape;
        ~~: send a '~' to the remote machine;
        ~%take from [to]: copy file "from" to local machine;
        ~%put from [to]: copy file "from" to remote machine.
```

To execute _call_ you will have to be logged in on some machine. If you are not, you can login as "remote." Instead of a shell you get a program that asks you for the machine you want to login on, and then executes _call_.

So each terminal is effectively connected to each machine. At the moment, if you inspect what each user is doing in our department, you will notice that half of them are executing _call_. It is useful because most of our machines are dedicated to one or two specific projects, and most of the faculty members are working on projects on different machines.

3.3. Remote execution

Many times you just want to execute a simple command at a remote machine without going to the trouble of logging in; _e.g._, you want to know if you are still in the top 10 of your favourite game on a certain machine, and if you are not you will have to login on this machine to fight for your place. Commands are executed on a remote machine with:

```
        rsh machine command
```

The output of the command is defaulted to the user's terminal, but can be redirected in the usual way, the input comes from "/dev/null." For example, "rsh A who" will give you a listing of the person's who are logged in on machine A.

It is now possible to run your programs on multiple machines. For example, if you want to run an _neqn/nroff_ job, you could run it on two machines as follows:

```
        (neqn file | to format)&
        rsh machine "from format | nroff -ms" > out
```

The _nroff_ output is redirected to the file "out" on the local machine. If you want to direct input to the remote command, and split standard output and error output, you could do something like this:

```
        rsh machine "from input | command | to output" >&2 &
        from output&
        to input
```

This means: execute command at the remote machine, with input from the process "from input" and output to "to output." Locally a "from output" is started in the background to catch the standard output of the command; the standard input is sent to the remote machine with "to input." The error output is done by the rsh process. If you put all this in a shell script, you can execute a command as if it runs locally. In the special case that this command is "sh -i," you can almost work on the remote machine as if logged in there.

3.4. Other Useful Programs

Out of the basic elements of file transfer and remote execution many interesting programs can be built. In this section we will discuss the programs used most on our machines; all these programs are shell scripts. Many of these scripts call to and from, which need a unique identifier as argument; for this purpose, the program uniqport outputs a random string of printable characters, to used as argument to from or to. The presented implementations of the programs are slightly simplified.

For file transfer it is a nuisance to have to login on two machines; therefore, we made a shell script called rcp which transfers files from any place in the network to any other place. Its syntax is:

```
        rcp [machine1!]file1 [machine2!]file2
```

This will transfer the first file to the second. One can leave out the machine part if the file is on the local machine. An implementation, in which the machine parts are non-optional, could be:

```
        IFS=! port='uniqport'
        (set $1; rsh $1 "cat $2 | to $port")&
        (set $2; rsh $1 "from $port | cat > $2")
```

A program related to rcp is rcat, with syntax:

```
        rcat [-] [machine!]file ...
```

and obvious meaning.

An implementation of this command, with exactly one file argument, could be:

```
IFS=!
set $1
case $£ in
1) cat $1 ;;
2) rsh $1 cat $2 ;;
*) echo "Usage: $0 [machine!]file" >&2 ;;
esac
```

Here is another thing about programmers: they are nosy. They want to know where their fellow-programmers are logged onto, and what they are doing. For this purpose we created the programs rwho and rw, which give information about the whereabouts and actions of all person logged in on any machine.

In our department we have several different printers attached to several different machines. Some produce ugly output fast, others produce pretty output slowly. It would be nice to print a file on an appropriate printer, independent of the machine the printer is attached to, or the system the file is on. With the program rpr you can do the same as with lpr, but with the advantages of location independence:

```
rpr printer [file ...]
```

Its implementation, in a configuration having two printers on the machine called "tjalk" and one on the machine "klipper," is:

```
case $printer in
tjalk)   mac=tjalk   com=lpr ;;
pmds)    mac=tjalk   com=opr ;;
klipper) mac=klipper com=lpr ;;
*)       echo "$0: unknown printer" >&2; exit 1 ;;
esac
port='uniqport'
rsh $mac "from $port | $com" &
shift
pr -t $@ | to $port ;;
```

Each shell script was written in 1 to 15 minutes; the basic elements of our network utilities (from, to and rsh) have proved their strength.

3.5. Implementation

Now having described the communication programs and the shell scripts we have built with them, we will discuss how from, to, rsh and call are implemented; in particular, we will take a look at the servers needed. All these programs use transactions as communication mechanism.

The implementation of _from_ and _to_ is simple: _from_ acts as a server waiting for request to output data to standard output, _to_ acts as a client doing transactions requesting the _from_ process to output the data _to_ has read. The port used in the transaction header is just the identifier given as argument to _to_ and _from_.

To execute a command on a remote machine, a server is needed that awaits a request and executes it when one arrives. The _rsh_ command is nothing but a client process doing a transaction with this server, requesting a command to be executed, and awaiting a reply saying the command has been executed. The servers on the different machines listen to different ports; given a machine's name, _rsh_ knows the port to use*.

For remote login one also needs a server. Although the server for remote execution could be used for this purpose too, a new one is made. A simple-minded implementation of _call_ could be the following:

```
rsh machine "from input | sh -i" &
to input
```

The problem here is that the remote shell has pipes for input and output; for example, you can not do ioctl's, or send signals along pipes. Therefore, we installed a device driver implementing a "pseudo terminal." The job of the remote login server is to manage these pseudo terminals.

A pseudo terminal really consists of two devices: a master and a slave device. The master device can be opened by a process simulating the terminal by writing to it for terminal input, or reading from it for terminal output; the slave device just looks like a terminal device to UNIX. The master device is called "/dev/ptyXX," and the slave device "/dev/ttyXX." The slave device is put in "/etc/ttys" as the other terminals are, so a _getty_ process can manage it. The master device has two processes driving it: the first writing to it simulating the pseudo-keyboard, and the second reading from it simulating the pseudo-printer. These processes are just _from_ and _to_, so that the pseudo terminal can be controlled at the local machine. All the remote login server does when it gets a request, is pick a free pseudo terminal and start the _from_ and _to_ processes.

The client process _call_ sends a message to the server requesting for a pseudo terminal, sets the local terminal in RAW mode, and starts a _from_ and a _to_. The _from_ catches the output from the pseudo terminal, and the _to_ will send its input to the pseudo terminal. _Call_ just copies its input to

*The port is a function of the machine's name.

the _to_ process via a pipe, except for the lines beginning with a '~', for which it must do some local processing.

As an example of how this mechanism works, we will consider what happens when the user types a DEL character, with the intention to generate an interrupt at the remote machine. First, the DEL is read by the local terminal driver, but because it is working in RAW mode, it just passes the character to the reader: the _call_ process. _Call_ outputs it in the pipe, giving the DEL to the _to_ process, which sends it to the remote _from_ process; _from_ writes it to the controlling site of the pseudo terminal device. Now the DEL character is treated as if the pseudo terminal was an ordinary terminal where a DEL was typed in: an interrupt is sent to all the processes belonging to the process group of this terminal.

Although the characters typed in when executing _call_ pass through a pipe, are sent to and echoed by the remote machine, and thus sent over the network twice, they are sent back to the terminal fast enough to see only a delay in the exceptional case of a lost packet, when the corresponding character has to be retransmitted. All the network programs are fast enough to work with, even by impatient programmers; but their success is mostly because of the simplicity of usage.

_4_. _Performance_

In this section we will give some performance figures for the rates we achieve using _from_ and _to_. They were measured during the middle of the day, _i_._e_., many persons were logged in, of whom some were working. Running the tests on a single user system sometimes doubles the data rate, but these figures are not of any importance, since in practice the systems are always multiuser. On the other hand, the performance drops fast if the systems are heavily used. The rates, as shown in Fig. 1, are not bad compared to most other systems.

	VAX 750	PDP 11/44
VAX 750	25,000	15,000
PDP 11/44	15,000	10,000

Fig.1. Data transfer rates in bytes per second over ProNET from user process to user process. The VAX's run 4.1BSD, and the PDP's V7. The buffer size is 512 bytes.

When we made the buffer size 2048 bytes on the VAX's, we achieved a data rate of 90,000 bytes per second (without file I/O). Unfortunately we could not use this size in general as we could not enlarge the buffer size on the PDP's.

As it does not matter where you run the network software, you may also run _from_ and _to_ on the same machine. The rates we achieve now are in Fig. 2. As these rates are the same as when running _from_ and _to_ locally, we may conclude that ProNET is not the bottleneck, but either the protocol or UNIX. Since our protocol is light weight, it must be UNIX. Indeed, when we look at where the most time is spent, it is in copying the user buffer to a kernel buffer, and in setting the timers.

VAX 750	PDP 11/44
25,000	10,000

Fig.2. Local rates. _From_ and _to_ both run on the same machine, and do not use ProNET.

References

Mullender84. Mullender, S. J. and Renesse, R. van, "A Secure High-Speed Transaction Protocol," _Proceedings of the Cambridge EUUG Conference_, (1984).

Saltzer80. Saltzer, J. H. and Pogran, K. T., "A Star-Shaped Ring Network with High Maintainability," _Computer Networks_, (4) pp. 239-244 (1980).

Tanenbaum81. Tanenbaum, A. S. and Mullender, S. J., "An Overview of the Amoeba Distributed Operating System," _Operating System Review_. Vol. **15**(3) pp. 51-64 (July 1981).

SNDAWK - A SIGNAL PROCESSING LANGUAGE

Dan Timis

I. R. C. A. M. 31, rue St. Merri, 75004 Paris.
(mcvax ! ircam ! timis)

ABSTRACT

Signal processing and sound synthesis with the **UNIX** system often use pipelines of programs that perform specific treatments such as filtering, changing the sampling rate or the gain, etc. on binary floating point samples. Users who want to use their own algorithms must spend, for even a very simple treatement, time and energy to write, compile and test a program in **C** or **FORTRAN**.

sndawk (sound *awk*) provides a signal processing programming language that is simple to learn and easy to use. Inspired by the well known pattern scanning and processing language *awk*[1], it follows much of its syntax as it includes much of the syntax of **C**[2].

1. Why *sndawk* ?

There are essentially two kind of sounds to process: recorded sounds (often natural sounds) or synthesized ones. Recording sounds with a computer means converting an analog signal into a digital one and storing the samples in a ***sound file***. In order to play a sound, one must convert the samples of a sound file into an analog signal.

Programs which synthesize sounds can write samples on a pipe or store them in a sound file. We can think of the following scheme:

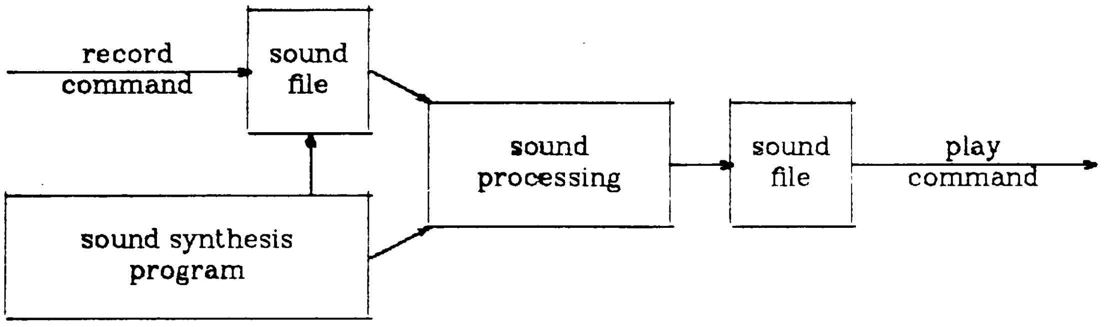

fig. 1 - Sound processing chain

At **IRCAM**[†] we use a real time sound file system, *csound*[3], and many programs performing standard signal processing algorithms on binary floating point samples, written at **CARL**[‡].

In order to read a sound file, to apply a gain to the samples and to store the result on another sound file, one must type the command line:

```
sndin foo |gain - 6dB | sndout bar
```

Programs like *gain*, *filter*, *reverb*, etc. are written in **C** and are easy to use. When processing a very large amount of samples, the user can put the command in the background and, in the meantime, do something else.

Applying a non-standard algorithm using the **C** language means:

- **writing** the program (or modifying an already existing one),
- **compiling**,
- **testing**,
- and finally **running** the program.

This could be a good solution if we would like to use the program many times. Otherwise, an interpreted language, with the program written in the command line (like *awk*), is a better solution. One could apply a gain using the command:

```
sndin foo | btoa | awk '{print $1 * 0.5}' | atob | sndout bar
```

where *btoa* (binary to arabic) converts binary floating point samples into arabic numbers (as text), and *atob* converts arabic to binary. Of course, a pipeline with a lot of conversions plus an interpreted program processing text, cannot be very fast. But to type such a command line for a non-standard algorithm, and to wait the result, is very often much faster than **writing**, **compiling**, etc., a program in **C**.

The syntax of *awk* is not well adapted to signal processing and the use of text instead of binary floating point could not provide satisfactory speed in execution. A command line like:

```
sndin foo | sndawk 'print $ * - 6dB' | sndout bar
```

is a better solution (of course, assuming the algorithm used is not standard and there is no other already existing program performing it).

†Institut de Recherche et Coordination Acoustique/Musique
‡Computer Audio Research Laboratory (UCSD)

2. Usage, structure and syntax

sndawk will read binary samples when its standard input is a file or pipe, and text when the input is a tty, the output following the same rule. Like its model ***awk***, the program can be written as an argument in the command line or in a file:

sndawk -f file

Pattern matching on binary samples is much less meaningful than on text. While ***awk*** searches files or standard input for patterns and performs actions upon lines or fields of lines, ***sndawk*** simply performs specified actions upon binary floating point samples present on standard input. So the program structure:

pattern { ***action*** }
pattern { ***action*** } ...

of ***awk*** becomes:

action
action ...

for ***sndawk*** (braces become useless).

Actions (or statements) will be applied on each input sample (or group of input samples). The special patterns **BEGIN** and **END** will allow capture of the control before and after any input sample is present on the standard input. These are optional, therefore the grammar is:

<Program>	::=	<BEGIN section> <statement section> (or **main loop**) <END section>
<BEGIN section>	::=	**BEGIN** <statement> | **empty**
<statement section>	::=	<statement list> | **empty**
<END section>	::=	**END** <statement> | **empty**
<statement>	::=	"a **C** like statement"

fig. 2 - *sndawk* **grammar**

Flow-of-control statements **if-else**, **while**, **for**, and statement grouping between braces, as in **C**, are provided. Statements are separated by semicolons, newlines or right braces. The *awk*-like statements **print** and **printf** will respectively put binary floating point values (when output is not a tty) or text on the standard output. Thus:

```
printf "%f\n" $ + 1
```

will force text to be written on the standard output, even if it is a file or pipe. A file can be written or appended using the > or the >> notation. Other **UNIX** commands or **CARL** programs can be called. Thus:

```
print $ | "sndout foo"
```

will write samples in the *csound* file **foo.**

3. Variables and expressions

Expressions are similar to those used in **C**. Arithmetic operators including **%** (modulo) and ^ (power), relational and logical operators together with operators ++, --, +=, -=, etc. can be used in expression lists, multiple assignments and can be grouped with parentheses. Arrays and simple variables need not to be declared and are initialized to **0**. There is only one type: **32-bit floating point**. Even relational expression are floating point therefore given:

```
x = a < b;
```

x will be **1.0** or **0.0** according to the value of the relational expression (**true** or **false**). Many functions of the mathematical library are provided:

fabs, floor, ceil, exp, log, log10, sin, cos, tan, asin, acos, atan.

Some special variables like **SR** (sampling rate - default 16000 sample per second), **NS** (number of the current sample - incremented automatically) and **TM** ($\frac{NS}{SR}$ time in seconds) can be used.

Arrays are considered to be sampled functions so **interpolation** or **extrapolation** are used when addressing them. A special variable (**IP**) can be used to choose the degree of the polynomial interpolation (default linear). Of course, interpolation is not used when an array element is the left value of an assignment. While **a[2.25] = 0.5;** (equivalent to **a[2]** = 0.5;) will be accepted, **a[-10]** = 1; will produce an error. Statements like **print a[2.25]**; or **print a[-10]**; are correct and sensible. Thus with:

```
IP = 5;
a[0] = 0;
a[1] = 1;
a[2] = 0.5;
a[3] = 0.25;
a[4] = 0.125;
a[5] = 0;
```

we could think of the following envelope function:

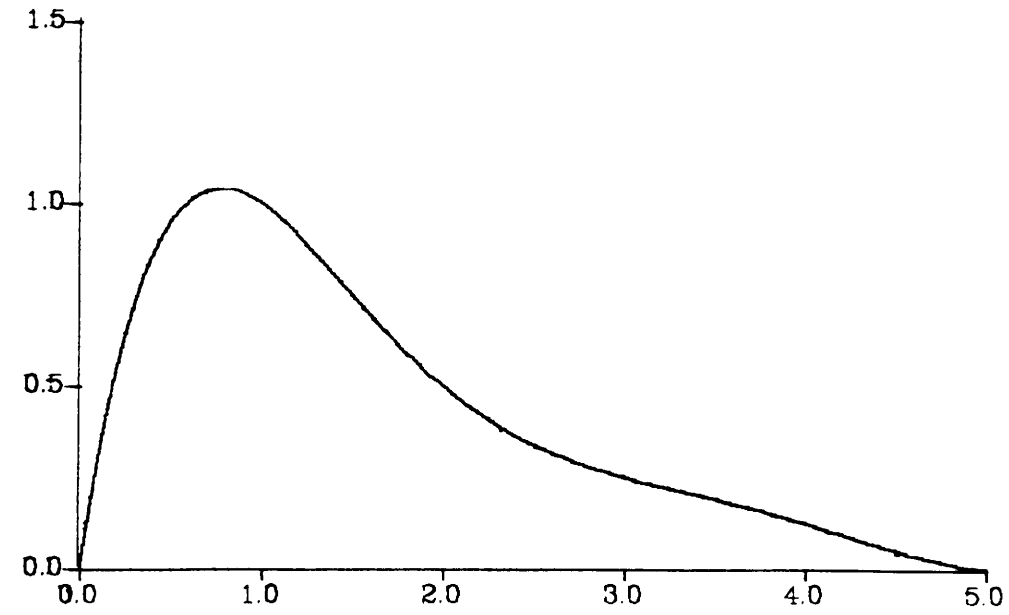

fig. 3 - An envelope function

Some *post-operators* are provided. Thus **-6dB** will be **0.501187**, **2sec** will mean **32000** (with **SR** = 16000) and one could write **print sin(100Hz);**.

$$(expr)\mathbf{K} = 1024 \times (expr)$$

$$(expr)\mathbf{k} = 1000 \times (expr)$$

$$(expr)\mathbf{dB} = 10^{\frac{(expr)}{20}}$$

$$(expr)\mathrm{sec} = \mathbf{SR} \times (expr)$$

$$(expr)\mathrm{Hz} = \frac{2\pi\mathbf{NS}}{\mathbf{SR}} \times (expr)$$

fig. 4 - Post-operators

4. Input and output

Of course some would like to use a filter:

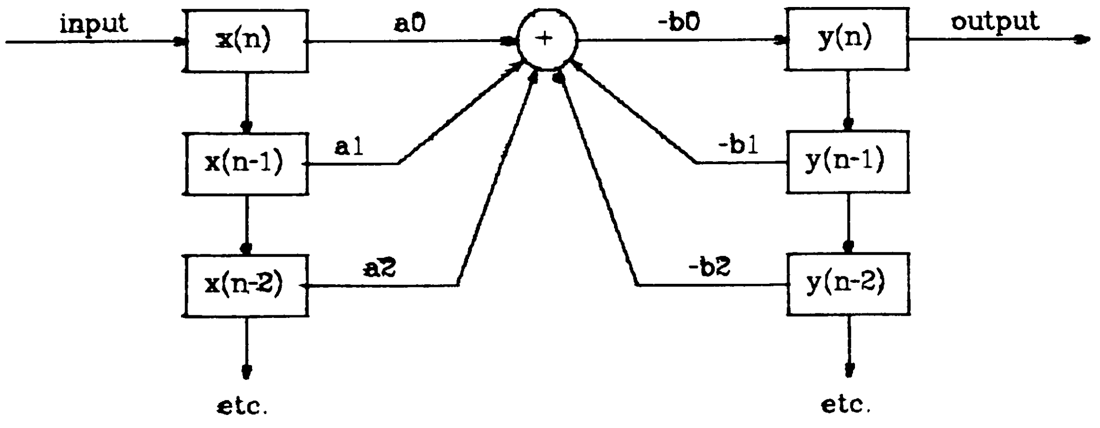

fig. 5 - A filter

and would want to keep some of the input and output samples. Others would like to process multi-channel sounds (stereo, quadrophonic, etc.):

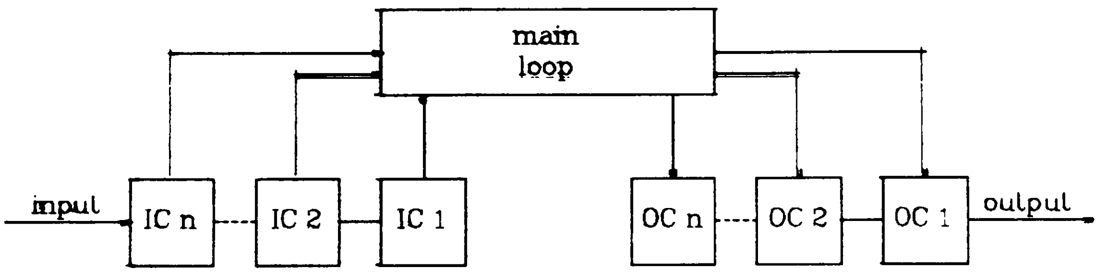

fig 6. Multi-channel processing

and will want to read and write more than one sample each time the **main loop** is executed.

Two multi-channel delay lines, one for the input the other for the output are provided. Special variables can set, in the **BEGIN** section of the program, the length and number of channels for input and output. **ID** is the length of the input delay (**OD** for output) and **IC** is the number of input channels (**OC** for output).

The current sample is indexed by **0** and the first channel is indexed by **1**. The default is no delay (only current input and output samples) and one channel. The input is denoted **$**[*delay*:*channel*] and the output **&**[*delay*:*channel*]. Thus **$[1:2]** means the previous group of input samples, channel 2. When the expression for *delay*, for *channel* or both are missing, the respective default value is taken. Thus **$** is equivalent to **$[0:1]**, **&[:2]** to **&[0:2]** and **$[1:]** to **$[1:1]**.

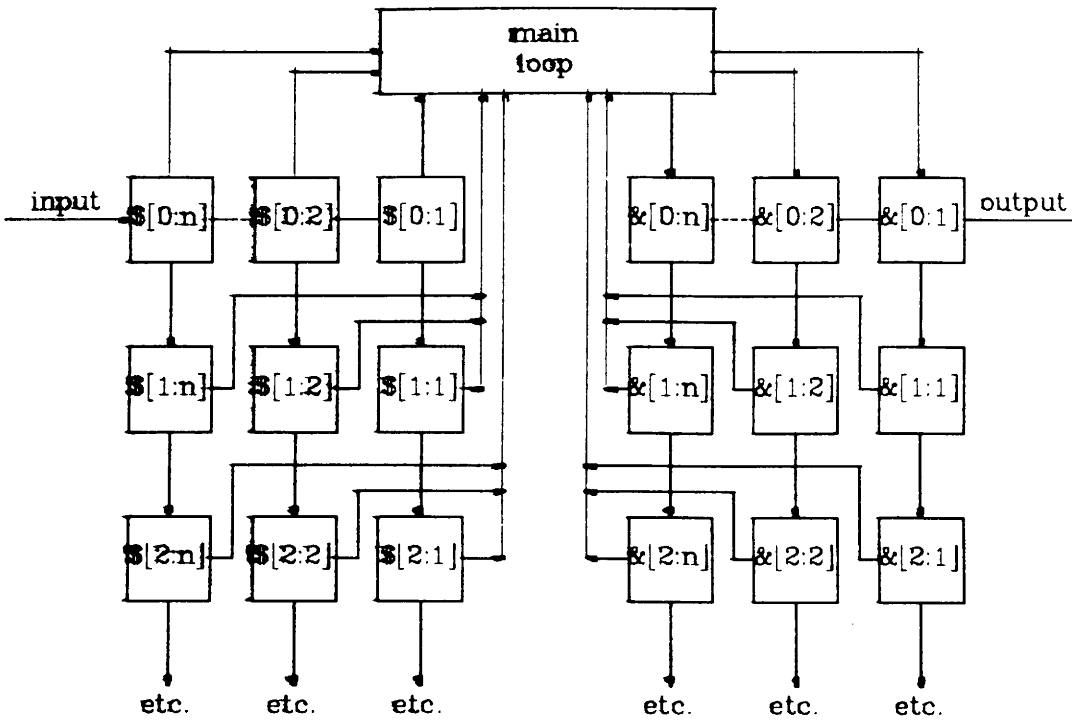

fig. 7 - Multi-channel input and output delay lines

5. Design and implementation

The lexical analysis of the program is performed by *lex*. The syntactic analysis is done by an syntactic analyzer generator of the author's design. The program is translated into a tree which is directly interpreted.

After executing the **BEGIN section**, the special variables **IC**, **ID**, **OC** and **OD** are consulted and the input and output delay lines are built. A number (equal to **IC**) of samples are read, then after one execution of the **statement section**, if there was no **print** or **printf** statement; output samples are written. Before repeating the cycle, delay lines, which are circular lists, are rotated. Users must specify which value is to be assigned to the current output and can use either the **print/printf** statement or the default output.

The functions **getfloat()** and **putfloat()**, from the *procom*[4] library are used to read and to write samples, providing full compatibility with other **CARL** programs.

Tree execution is not very fast for the large amount of data required by sound processing but the conciseness of the language and the absence of the compiling phase make it very comfortable to use. A simple test for speed shows that if a **C** program is faster than an equivalent *sndawk* one, *awk* is much slower. The test was made for the following command lines:

```
gain 2 < data1 > data2

sndawk '& = $ * 2' < data1 > data2

awk '{ print $1 * 2 }' < text1 > text2
```

data1 contained 16000 floating point samples and text1, 16000 numbers, one number per line with no space.

	real	user	sys
gain	6.0	1.6	0.7
sndawk	18.0	11.0	1.1
awk	4:23.0	1:46.9	8.1

fig. 8 Speed test

Some simple examples of programs are given in the appendix.

References

1. A. V. Aho, B. W. Kernighan and P. J. Weinberger, *Awk - A Pattern Scanning and Processing Language*, Bell Laboratories, Murray Hill, New Jersey (1978)

2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978)

3. D. G. Loy, *Introduction to the csound file system*, Computer Audio Research Laboratory, UCSD, La Jolla, California (1982)

4. D. G. Loy, *Procom - interprocess sample data communications facility for UNIX*, Computer Audio Research Laboratory, UCSD, La Jolla, California (1982)

Appendix

IMPULSE RESPONSE FOR A FILTER

```
impulse 100 | sndawk 'BEGIN OD = 2; & = $ + .99 * & [ 1 : ] - .9 * & [ 2 : ]'
```

STEREO TO MONO

```
BEGIN IC = 2; & = .2 * $ + .8 * $ [ : 2 ]
```

REVERB

```
BEGIN { SR = 32000; ID = 0.5sec; }
& = 0.5 * $ + 0.1 * $ [ 0.3sec: ] + 0.05 * $ [ 0.5sec: ]
```

FREQUENCY MODULATION

```
print sin (( 200 + 10 * sin ( 5Hz )) Hz )
```

MELODY

```
BEGIN {
                do = 261.62;
                re = 293.66;
                mi = 329.62;
                dur = 0.8;
}
if ( TM < dur ) print sin ( do Hz );
else if ( TM < 2 * dur ) print sin ( re Hz );
else if ( TM < 3 * dur ) print sin ( mi Hz );
else print sin ( do Hz );
```

APPLY ENVELOPE

```
BEGIN {
                IP = 5;
                env [ 0 ] = 0;
                env [ 1 ] = 1;
                env [ 2 ] = 0.5;
                env [ 3 ] = 0.25;
                env [ 4 ] = 0.125;
                env [ 5 ] = 0;
                maxenv = 5;
                totaldur = 3;
}
print $ * env [ TM / totaldur * maxenv ];
```

Published by the European UNIX® Systems User Group,
Owles Hall, Buntingford, Herts SG9 9PL.
Tel: Royston (0763) 73039.