More Taste: Less Greed?
or
Sending UNIX to the Fat Farm

C H Forsyth

Department of Computer Science
University of York
Heslington
York YO1 5DD
England

+44 904 432753
forsyth@minster.york.ac.uk

ABSTRACT

You (like us) have 80 to 90 Sun 3/50 machines with 4 megabytes of memory. You have been given some optical discs containing System V.4. Which can you least afford to discard? Things are getting out of hand. Soon, 32 megabyte Ultrasparks will be needed just for us to start the window system in under a minute.

For UNIX, now in middle-age rotundly recalling its sprightly youth, mere 'tuning' will not cause that heavy code to slip away. We need to reconsider and re-implement the system interface periodically, to take account of changes in its environment. We must be willing to *throw things away*, discarding parts of the older implementation completely, rather than corrupting clean new mechanisms to approximate the mistakes of the past.

To illustrate this thesis, I shall discuss work I have done on SunOS 3.5 to reduce its size and complexity. The virtual memory system has been replaced by a simpler one using ideas from the EMAS system and elsewhere. A stream IO system in the 8th/9th Edition style has been added, replacing the old character IO system, pipes, and UNIX domain sockets. I have also made some preliminary forays into the Virtual File System interface.

1. INTRODUCTION

Many papers describe various ways of dealing with the complexity of 'programming in the large'. There are fewer that take a large program apart to try to find out why it is large, how it grew, and whether it can be made rather smaller. The commonly available versions of the UNIX system have been growing relentlessly with each new release and each new feature; the 'standard' UNIX kernel is now a moderately large program, surrounded by very large applications. I have watched it become much harder to understand, maintain, and modify. This troubles me directly (since I provide system software support), and so over the past few years I have made various attempts, very much part-time, to recode or rewrite parts of the system to be smaller and simpler. The work began on a VAX with the 3rd Berkeley Distribution, and continues on the Sun 3 series with SunOS 3.5. The changes have been growing more extensive with time.

Here, I shall pick examples from various parts of the system to show how certain designs match the structure of UNIX well, and thus make things easier; and conversely, where there is a serious clash, the implementation becomes much more complicated and the facilities less useful. It can also be less painful in the end to replace some older things completely, rather than pile hack upon hack. I claim it is a mistake to standardise (or to have standardised) certain parts of the system interface in their current form.

Background

The UNIX kernel I ran until 1987 on our VAX machines was a 're-port' of 7th Edition to the VAX, but it included: a recoded 3BSD paging system, many devices, an X.25 interface, and my own implementations of ideas from 8th Edition UNIX (which was not generally available), specifically the file system switch [Wei84], **/proc**[Kil84], and a 'stream' IO system [Rit84]. The latter included pseudo-terminals, the message and connection disciplines, and stream drivers and modules that provided some of the Ethernet and Internet protocols: **deuna**, **arp**, **ip**, and **udp**. There were two peculiar special files: **/dev/time** and **/dev/wait**. By the standards then prevailing the kernel was acceptably small:

```
/unix: 95280+20400+146252 = 261932b = 0x3ff2cb
```

(All kernel tables are included in the data and bss sizes; IO buffers are not.) Starting with ‘layers’ [Pik83] code by Simon Kenyon, Nigel Roles wrote an implementation of Pike’s window system **mpx**[Pik84] that was good enough to run programs such as the text editor **jim** for the Atari ST. We had a promising environment.

In 1987, however, we decided for financial reasons to switch to Suns (discless 3/50s supported by file servers). I was disconcerted, not to say disheartened, to find that the Sun kernel code was about 390 kbytes, even properly configured (as distributed it was 490k). It had notionally been built on 4.2 BSD, but the changes to support the Virtual File System and ‘vnodes’ [Kle86] were extensive, although the character IO system and socket systems were largely unaffected.

Investigation

I decided to discover why the system was so much larger and (apparently) complicated than previous UNIX systems. I thought that a good way to do this was gradually to replace parts of it, and compare the size and complexity of the old and new versions. Rewriting code forces one to (re)consider all the problems the original author was trying to address. Furthermore, a ‘paper’ redesign would not only be unconvincing, it would do little to provide the facilities, particularly streams, that had been lost by discarding the VAX system.

A significant constraint was that the system must run SunOS 3.x binaries: we had software such as Framemaker for which we had no source, and programs would also need to be exportable without change.

Excluding devices, the larger components of the kernel were: the virtual memory system; the window system; the 4.2BSD File System; basic kernel functions including fork, exec, signals, and scheduling; the network file system; sockets and buffering, including pipes; virtual file system support; and the network software. I decided to look at a few of them more closely, taking advantage of the experience gained on the VAX.

Some degree of improvement was ultimately made in all areas except the Fast File System and the network side of the socket code. (Eventually both might be replaced.) The CPU scheduling code was replaced by a variant of a simple yet flexible scheduler developed at the University of Maryland [Str86].

The new system has been in use in various forms for more than a year on several Sun 3/50s and one 3/160 in the Department. Depending on configuration, it is currently about 120kbytes smaller than the old (the Sun window system still takes about 60 kbytes in the current kernel). There is still much scope for improvement, but even now the system does more, uses memory more efficiently, and is (I claim) better structured.

In following sections I shall discuss the changes to the paging system, changes made or currently being made to the virtual file system, the installation of the stream IO system, and the implications of all that on kernel implementation and applications.

2. MEMORY MANAGEMENT

The SunOS paging system was completely replaced. The new system uses a ‘working set’ paging scheme with a local paging policy, instead of the previous global one. Some implementation ideas, and also extensive justification for this approach, were found in the published descriptions of the EMAS system [Ree81, Ste80, Ste80]. Two other virtual memory systems, by Richard Miller [Mil84], and by a group at Gould [Akh87], both written for versions of UNIX, were also examined closely (our various implementations differ). My goal was to provide a reasonable implementation of the basic UNIX process model on a machine with virtual memory, not to add new primitives; I wished the result to have very few moving parts. The use of the EMAS scheduling algorithm is perhaps the most novel aspect.

2.1. The original paging algorithm

The Berkeley paging system uses a ‘clock’ algorithm to approximate LRU page replacement across the system. This has undergone some adjustments with time, but the basic scheme is as follows. (See [Lef89] for precise details.) When a process requires a new page it simply takes a free one from the global page pool as soon as one is available. Pages are made available when processes terminate, but also by the intervention of a system process, the ‘pageout daemon’. There is a big array, the ‘core map’, with one entry for each usable page of memory. The pageout daemon cycles through the array at a rate determined by some scheduling parameters. Free pages are skipped. Any other page is examined to see if it can be discarded.

The daemon checks the page table entry for the page to see if the ‘page used’ bit is set. If not, the page can be made available. If it has been modified, it must first be written out (asynchronously). If the page has been referenced, the daemon resets the ‘used’ bit and proceeds to the next core map entry. If the page has not been touched the next time round, it will be freed. On systems with large memories, a complete cycle can take some time, and so recent versions have the daemon cycling two pointers through the map: one clears ‘used’ bits; the other reclaims pages.

In principle, this is a simple and elegant technique. It has been applied quite effectively in a controlled environment in a new implementation of file cacheing in UNIX using virtual memory [Bra89]. There are, however, several problems in practice with the algorithm when applied to UNIX processes, at least in its SunOS 3.5/BSD 4.2 version: the core map must keep a back reference to a page table entry, a process’s working set size is not always accurate, and the memory scheduling algorithms are elaborations of the early ‘swapping’ algorithms.

The daemon must be able to locate ‘the page table entry’ for each page. The core map entry for a physical page must therefore point back to the page table entry that refers to it, if there is one. Map entries for data and stack pages store a (process, page) pair. Pages of an executable program, though, can be referenced by several such entries, which must all be found. The solution adopted is to store (text-table, page) in the core map. Processes using a given executable are linked in a chain from its text table entry. Whenever one of the processes’ page table entries changes, owing to page in or page out, all others must change to match it. The pageout daemon looks round the list too, in order to calculate the union of used/modify bits for all processes referring to the page.

When the page daemon takes a page, it invalidates the page table entry but leaves a reference to the page (the page frame number), so that should the process soon fault on that page it can reclaim the page directly from the free list. Before any process is given a free page, the memory allocator must therefore clear any reference to it, using the data in the core map in much the same way as the pageout daemon. Just to liven things up, in 4.2BSD the network system can claim pages at interrupt level using the memory allocator, so that a process’s page table entries can be changed by an interrupt! (In practice this is not a problem provided care is taken at the kernel main level, but it is something else to have to reason about when checking the code.)

To reduce overhead, the old pageout daemon does not run until the amount of free memory drops below a specified value. It then cycles round the core map at a rate that depends on the amount of memory left. If it cannot free enough memory quickly enough, the memory scheduler (swapper) is started. The swapper synchronously writes to disc clumps of the modified pages from the whole image of each process it selects.

2.2. Simplified paging

Some of the complexity in the old paging system results from using a global paging policy, which causes pointers to go ‘the wrong way’ for UNIX, where pointers elsewhere almost invariably go from a process to a shared resource which is given a reference count. For instance, when sharing executable code, pointers go from the process to a ‘text table’ which has the reference count. When sharing open files, pointers go from a per-process open file table to a reference-counted system open file table (which has the seek pointer), which points in turn to the file’s in-core ‘inode’ (or vnode), which again has a reference count. In contrast, the solution adopted for sharing text pages is adequate in practice, but becomes messier when data pages are to be shared (for instance, in some implementations of ‘fork’).

Pointers, then, should go from the process to memory (the shared resource). Pages can be shared easily if there is a reference count for each page. In the new system, a process releases its own pages when it no longer requires them. Periodically, each process examines its page table entries and discards pages that have not been used since the last such scan. It resets the ‘used’ bit on the remaining pages. Over time, the process tracks its own working set.

When a process discards a page during a working set scan or a complete page out, the page is put in either a free page or modified page list depending on whether it has been modified since being read in. Pages on the modified list are written to disc by a separate system process (the new ‘pageout daemon’). It does not write pages out immediately, but only when the modified page list grows too long, or when the free list length drops below a specified level. This increases the chance that the page can be reclaimed by its process without IO overhead. Pages are returned to the free list after they have been written out. The new pageout daemon is a simple loop: it waits for work, collects pages from the modified list, and writes them out.

By itself, use of a working set scan in each running process is not enough to prevent thrashing, since the combined working sets of all such processes might not fit in memory. The collection of processes in memory must be subject to some global constraint. The most obvious constraint to apply is that before a process is loaded there must be enough store to hold its working set. This typically changes with time, however, and so a scheduler is required that can adapt to these changes.

The EMAS system has an elegant scheduling system [She74] that was chosen for use in our system. Each process in EMAS is assigned a category, a single number that captures its current resource requirements, and also determines its CPU and memory priority, working set scan frequency, and CPU quantum size. When loaded by the scheduler, a process is given a page and CPU time allocation determined by its category. It remains loaded and in the CPU despatcher's queues until it exhausts either allocation or goes to sleep. The process must then submit itself for rescheduling.

The scheduler refers to a resource model for the system, encoded in a *category table* which is indexed by process categories. The table can be viewed as a little state machine with transitions from category to category. There are four transitions from each category, labelled 'more memory', 'more time', 'unused memory', and 'unused time'. The scheduler follows the appropriate path to obtain a new category that suits the process. In this way the scheduler quickly tracks the character of a process as it changes.

The rescheduled process may remain loaded if it is runnable, its new allocation fits in the available memory, and no higher priority process is waiting. It must otherwise page itself out, and join a priority queue to wait for memory. A process sleeping on an external event must likewise page itself out whilst waiting unless the memory it occupies is not in demand. When it wakes, it joins the appropriate memory queue. A memory scheduler process cycles through the priority queues using an auxiliary priority ratio table, loading each process in turn as memory becomes available. (See the EMAS papers for details. Wilkes [Wil75] includes the EMAS scheme in an interesting discussion of paging viewed as a problem in control theory.)

The notion of an 'external event' is hard to sustain in modern UNIX systems. Instead, the priority of a sleep† is assumed to say something about the likely speed of response. When a process sleeps at high priority, it records the unused part of its quantum, adjusts its CPU priority (see below), and enters the sleep queues as usual. A process sleeping at low priority first checks with the memory scheduler before entering the sleep queues. If its memory is required now, it will page itself out, as in EMAS, but otherwise it puts itself in a priority queue of processes that are 'idle' in memory.

The new memory scheduler has a simple loop: it checks the 'wait memory' queues in the order given by the priority ratio table, and once it has found a process, it waits for memory to become available, as lower priority processes page themselves out. It also watches for the arrival of idle processes, and when memory is needed causes them to page themselves out, starting with low priority processes, as soon as they have been idle 'long enough' (a scheduling parameter).

The original scheduler had the hardware clock interrupt increment a 'time' field in each process descriptor to decide how long a process had been in or out of memory, or sleeping. Berkeley moved this to a function called periodically by a software clock, but it still required a scan of all living processes. Instead, in this system, a process is simply time stamped when it enters any (FIFO) scheduling queue. (Currently the stamp is the time in ticks since system startup.)

2.3. Improving UNIX implementation

The previous section gave an overview of the old and new paging systems. Let us look now at some areas in more detail, to compare the complexity of the implementations.

- **Fork.** The `fork` call makes a copy of the caller's memory image. As many have observed, this is often wasteful, since in most cases the `fork` precedes an `exec` which throws the copy away! Paging everything in to make the copy will also disrupt tracking of the working set. Fortunately, copying is just the specification; we are free to implement it in the most efficient way. Many have suggested (and several have implemented) sharing memory between parent and child until one of them writes to a page; the write is intercepted, the page is copied, and the write is completed in a private copy of the page. On some machines copy-on-write is hard or costly to implement, but copy-on-reference can be used instead and should be nearly as effective in the most common case (speeding up `fork` before `exec`).

Berkeley nevertheless implemented `fork` by complete copy, and added a new system call `vfork` instead. The parent passes its page tables to the child. The parent is suspended, marked 'no VM', and linked to the child, which is marked 'in vfork'. The child process returns from `fork` and continues to run in the parent's memory until `exec` or `exit`, at which time it wakes its parent, and waits. The parent grabs its memory

† This refers to the kernel sleep/wakeup synchronisation system, not to the C library function.

back, resets some pointers, and allows the child to proceed. Several parts of the system must check for 'in vfork' (including the terminal handler!). The system calls `exec` and `exit` must synchronise with (the parent's) `fork`, and the pageout daemon must follow the vfork links to find the current user of the page if its true owner is marked 'no VM'. Thus `fork` is moderately expensive, and `vfork` is rather odd.

The new system does use copy on write. The `fork` call copies the swap maps, incrementing the swap reference counts; copies the page tables setting 'copy on write' state and read-only access; and increments a core map reference count for valid pages (a page will not be put in the free list until the reference count goes to zero). There can then be two kinds of page access faults. On a 'write protect' error, a page that is no longer shared can have its permissions restored; a shared page must be copied. In either case the page becomes private to the process. (A private page also results from a 'page missing' fault if the page is 'fill on demand'.) Including the code to implement reference counts on swap pages, no more code is required to implement copy on write in this system than `vfork` took in the old.

• **Page theft.** In the original system, the pageout daemon wanders round the core map stealing pages. Each time, it must check the state of the process owning the page. If it is in a safe state, it temporarily maps in the process's kernel data area to access the swap maps. If the page being stolen had been modified, the daemon mapped it into its own VM and started asynchronous IO.

In the new system, nothing happens 'asynchronously' to a process as regards the paging system, since it either discards its own pages or must queue itself to allow the memory scheduler to do so. It simply does neither if it is in an inconvenient state, or if scheduling policy does not yet require it. The new working set scan routines `pagediscard` and `refscan` come to 51 lines including some optional event tracing calls (except when a page must be freed, no functions are called during a scan), but many more lines of code have been removed.

• **Reclaiming pages.** In either system, a process might lose or discard a page, then require it again before it has been taken from the free list for use elsewhere. As mentioned above, the old system leaves the page table entry non-zero to allow reclaim, but clears it as soon the page is taken for use elsewhere. It also has a hash table, in which pages can be found using the pair (source-vnode,block) as a key.

In his system, Richard Miller [Mil84] had an array of disc block addresses, one per memory page, each entry containing either zero or the current swap block address of the page. When a page table entry was invalidated, the page frame number was left non-zero. Before reclaiming a page, the system checked that the swap block address matched that expected by the current process. No action was necessary if the page was claimed by another process. A similar scheme is used here (the swap address is kept in the core map, for use by the pageout daemon). There is currently no need for a hash table. Each shared image (text) entry provides a 'master page table' that tracks the location of executable pages, which might page out from one frame and page in elsewhere some time later (but which should still be shared).

• **Swapping.** A swap operation in the old system writes out all the modified pages in a process. There are special support routines that find contiguous regions of dirty pages that are then split to fit in contiguous regions of swap space. There are some others that, during page out, find modified memory pages that are adjacent in virtual memory to the one selected by the page out daemon.

In the new system, a swap is just a complete process page out, which simply frees the process's memory; clean pages go on the free list, and dirty ones go on the modified page list. If dirty page table entries are adjacent, and they are freed in sequence, and at the same time, their associated pages will be adjacent on the modified page list. The core map entries for the pages have the swap addresses attached to assist page reclaim. The pageout daemon simply checks adjacent modified page list entries to see if the swap blocks are adjacent and if so writes them out with one IO request. This gives the effect of a 'partial swap' at little cost. The same effect is obtained by a process that releases adjacent dirty pages during a working set scan. The cost of a bad scheduling decision (eg, a sleeping process is swapped and immediately wakes up) is reduced, since a process can reclaim pages from the modified page list before they are written. (The per-process data and page tables are paged out and reclaimed in the same way.)

• **User control.** The use of working sets should help to address some of the problems mentioned by Rosenthal [Ros89], since the scheduler can more accurately track a process's memory requirements. Even an overenthusiastic process can be kept within bounds, paging mainly against itself, so that on a single-user machine for instance, a text editor running in a windowing system at the same time as such a process has a better chance of remaining responsive compared to the old system.

Scheduling policy can be controlled in several ways, most simply by changing the category table, although Stephens et. al. [Ste80] do warn that the effects of doing so are not always easy to predict, and might require experiment. The scheduling parameters and tables are accessible through special files in the directory

`/dev/sched`, making it easy to give workstation users control over the allocation of resources to processes on their machines. Windowing processes can be given a set of categories disjoint from the others, just as EMAS provided different sets of categories for interactive and batch processes.

2.4. Summary

The new paging system is just over half the size of the old one. I would claim that the new one is easier to reason about, since the various processes involved interact only when each is in a well defined state. They co-operate: a process does not barge into another's memory without knocking.

3. FILE SYSTEM TYPES

The first 'file system switch' appeared in the 8th Edition. Several interesting file system types were implemented using it: the network file system **/n**[Wei84] and the process file system **/proc**[Kil84]. Others have followed suit: the Generic File System [Rod86], and the Remote File System [Rif86] use (somewhat more elaborate) file switches. Sun developed the Virtual File System (VFS) and vnodes [Kle86].

I had implemented my own impression of the 8th Edition file system switch on our VAX. ('My own ...' because very few details had been published about the 8th Edition switch.) When we then moved to Suns, I took several of the older file system types, such as **/proc**, and tried to adapt them to work in the VFS/vnode system. I found it harder to use: it did no work for me! I did produce an acceptable solution to my immediate problems, but the VFS interfaces remain unsatisfactory. Closer study revealed some confusion and limitations in its design and implementation. To see why, we must look a little closer at what these systems do.

3.1. A simple UNIX file system switch

Adding a file system switch in the 8th Edition style to the 7th Edition system is fairly easy to do. A value is added to each mount table entry that tells which type of file system is mounted there. That value is used as an index into a table of file system operations, and the parts of the mount table entry used only by the disc file system are moved elsewhere. I found that extensive changes to the kernel source code could be avoided by keeping the `inode` as the 'generic file', so that references such as `ip->i_size` continue to work; the disc-specific information goes elsewhere. This seems reasonable: an inode *is* an abstract file. The interface to a file system is similar to the interface to character or block device handlers. Kernel functions such as 'read inode' are replaced by something like

```
void readi(struct inode *ip)
{
        (*filesw[ip->i_fs->m_type].f_readi)(ip);
}
```

The function bodies are renamed and moved into files associated with the disc file system type; suitable entries are made in the switch table. More work is required to add indirection to file name lookup and when managing the 'inode table' (iget, iput), and one might consider changing the calling conventions to hide from the file systems the structure of the per-process data area, but this should give the basic idea.

Within any file system, there have always been different kinds of files: directories, devices, sockets, FIFOs, symbolic links, and the 'regular' file. In 7th Edition, each file is tagged with a value telling its kind (IFDIR, IFREG, etc). The internal versions of read and write can be applied to any kind of file. Reading a regular file yields its data, but reading a symbolic link produces a file name, and reading a directory gives a list of (inode-number, name) pairs in some format. The internal version of write can also *write* an (inode-number, name) pair into a directory; it thus creates a link.

The same scheme can be used with the file system switch. The implementation of 'read' for the file system will look at the kind of file to decide what to do. Note that all file systems must agree on a standard format in which they will exchange directory entries with the kernel during read and write. This need not be related to the directory format that the file system uses internally. For example, **/proc** makes up directory entries 'on the fly' out of the contents of the kernel's process table. It is not necessary to add a special 'readdir' or 'getdirents' call.

Implementation in this way puts the file system switch below the bulk of the UNIX system call/file system interface code. The job of implementing UNIX semantics (pathnames, checking permissions, managing times, inode data structure locking, etc.) is mostly done above that. The file system writer can concentrate on the rewarding job of obtaining the data to make curious things look like files and directories to processes.

3.2. VFS and vnodes

At first glance the VFS/vnode system is roughly similar to the above. The VFS and vnode operations structures are different types. The file system independent part of the VFS system implements the file system name space, including mount points; it interprets path names. Once a particular file system has yielded its root vnode to the VFS code when it is mounted in the file system hierarchy, or when it is crossed at a mount point, it plays no further part in the interpretation of operations on files in the file system it represents. They are directed through the operation structure attached to the vnode; the vnode operation set is extensive.

Sun's system is avowedly 'object oriented': the group of function pointers linking to the operations is attached directly to the data structure or 'object' representing the file system (VFS) or the file (vnode). One might then expect each vnode to accept just the operations it implements: the operation structure attached to a 'directory' in some file system would accept 'lookup', 'rename', and other such directory requests, but its entry for 'readlink' would point to an error routine. Similarly, a 'symbolic link' would reject many other operations but accept 'readlink'. That is not what happens. In all the file systems in SunOS 3.5, every vnode in a given file system has the same operation set, which accepts any operation that is allowed by some kind of file in the file system. Each operation then checks a 'file kind' tag to see what sort of file it is, just like the other systems. (Indeed, it is sometimes checked twice: at the system call level, and by the vnode operation. The second check is not redundant, since the NFS server takes a shortcut, bypassing the first check by calling vnode operations directly.) As things stand, the operations are not really associated with the vnode at all, but with the collection of vnodes inside a particular file system, and the contents of the operation set are known statically. Putting the pointer in the vnode itself is just an optimisation.

3.3. Adding UNIX semantics

The VFS/vnode system is said to ''support (but not require) UNIX file system access semantics'' [Kle86]. This is accurate; the use of the phrase '*access* semantics' is telling. On a system call, the VFS code looks up parameter pathname(s) to obtain a vnode (or two). After that, unlike the older file system switch, VFS in most cases does little more than change the representation of its parameters and call the vnode routines. All the file system types that I wished to add were UNIX file system types, but it appeared each would have to maintain its own UNIX file system state and behaviour in the handling of user and group IDs, permissions checking, and the like. Scanning the code of the existing file systems confirmed this impression, and also revealed repetitions of significant code sequences. For instance, *specfs*, *nfs* and *ufs* all contained code to do IO through the buffer cache. *Nfs* and *ufs* separately implemented the checks of the `access` system calls. What a good way to get several (slightly different) copies of UNIX!

On the VAX system, the only file system types were UNIX ones, and so UNIX semantics had been implemented just once above the file systems by the system call level. I decided to provide a file system 'base implementation' that would support UNIX file system implementations. The kernel continues to operate on vnodes using Sun's interface and operations, although the VFS level was changed to provide vnode locking primitives, as a service to be used as required by the lower levels. Most of the real work is done by a new file system type.

The *unix* file type attends to the standard UNIX semantics for files, including maintenance of user and group IDs, protections, file mode, file times, size, provision of a basic set of values for `stat` ('getattr'), and a basic protection policy. It simplifies implementation of the *proc* file system, a *memory* file system (which allows a directory structure and files to be created in main memory), and some others to be described presently. It currently does not handle the *ufs* file system type (the 4.2 disc file system).

It also supports several types of anonymous vnodes, file types that lack names in the file system hierarchy: currently *pipe*, *fifo*, *socket*, and *config* are included. The *pipe* and *fifo* types provide vnodes for all stream IO (not just IO on pipes), the *fifo* subtype differing mainly in the open/close protocol. The *socket* and *config* types need more explanation.

3.4. Reconsidering socket and device switches

In 7th Edition, the system's 'open file' structure referred directly to the inode corresponding to that file. In 4.2BSD a level of indirection was added, to direct IO on a file descriptor either to the socket IO system or to the (inode) file system. The set of operations included read/write, close, ioctl and select, precisely a subset of the vnode operations added later, yet both mechanisms remained in the Sun system after the VFS was added at the next level down. The *socket* type provides suitable vnodes for IO on sockets, and has replaced the older two-tier mechanism. Having all files in the system accessible through the file system interface (inode or vnode) makes it more likely that applications can use any kind of file in most contexts.

A character device is represented by open, close, read, write, ioctl and select operations selected by indirection through a 'device switch'. Since vnodes provide a superset of these operations and the operation set can be unique to the vnode, perhaps the device switch is redundant.

The (experimental) *config* file system can potentially replace the device switches. It provides a device name space (hierarchy) derived automatically from the machine's configuration. Each file in it has its own set of operations that refer to the appropriate device handler routines. Multiplexors are represented by directories that generate channel vnodes on demand. (This might serve to eliminate major/minor device numbers from the system, removing the need to make them larger on big systems.) *Config* differs from *specfs* (Sun's special devices file system) in that it provides a mountable name space. *Specfs* makes a vnode for a device; *config*'s vnodes *are* the devices (or pseudo-devices). It is an experimental file system because I am uncertain how best to handle the file attributes. Originally in UNIX, all file systems were on disc, saving the permissions, ownership, and file times of device inodes over a reboot. Currently, because *config* is a virtual hierarchy, without persistent store, attributes on some files must be set by `/etc/rc` when the default attributes are not acceptable. This might turn out to be the best way.

3.5. Summary

I intended not so much to reduce the cost or size of the original implementation but rather to investigate its structure and provide a framework in which new (UNIX) file system types can be added easily and cheaply. The separation of responsibilities between the system call and vnode layers, the split between file system types and file types, and the location of error checking all seem unsatisfactory. It seems ridiculous to me that a UNIX system would provide so little support for UNIX semantics in the file system (the heart of UNIX) that such things as permissions checking should be implemented several times. I question the value of the current vnode system as a cost-effective tool when building UNIX file system implementations.

4. STREAMS

4.1. Streams (and STREAMS)

The stream IO system was originally developed by Dennis Ritchie for the 8th Edition. [Rit84]. It was later adopted by System V. [Dun86]. I implemented a stream IO system on our VAX in 1985, because we were installing various kinds of communications and networking facilities, and I thought stream IO would help to make sense of it all. Sockets would also have been available then, but streams had the advantage that it was much more than just a networking system. Stream IO is an elegant abstraction which is also extremely useful! What characteristics appealed to me?

- a stream is a full-duplex buffered channel into the kernel
- a stream contains a sequence of bytes punctuated by delimiters
- data is transformed only by processing modules on the stream
- data is read and written using read and write
- there are essentially no options

These help to make streams *neutral* in their application in the same way that any data can be written in any format to a UNIX file, or sent through a UNIX pipeline. By contrast, sockets provide options for datagrams, streams, sequenced packets, and a 'raw' interface: they clearly provide a *networking* interface.

There are two ways of dealing with networks in UNIX using streams: make the complex network look like a collection of simple streams [Pre85, Pre86], or change the stream system to look more like a network. AT&T took the latter approach when the stream IO system became ''STREAMS technology'' in System V. It acquired some new features: multiplexor drivers; the contents of a stream is now a sequence of messages, which are themselves sequences of bytes; and there are now options at the stream head, including IO modes. IO modes include 'message discard', 'message non-discard', and 'byte stream': they provide different ways of reading the data on the stream.

The result is something of a cross between the original stream system and sockets. The basic STREAMS implementation is consequently somewhat larger than mine: the support for streams, queues, and buffering takes 29,640 bytes in SunOS 4.0, compared to my system's 9,180 bytes. (The STREAMS figure does not include support added recently for independent 'out of band' message sequences within one stream [Rag89].)

4.2. Streams in SunOS

Last year, I installed a revised version of my streams system on our Sun system. Ritchie found that his kernel became appreciably simpler, smaller, and more useful when streams replaced some character devices and line disciplines. I rather hoped that might happen here. First, I replaced pipes by full-duplex streams, and FIFOs (named pipes) by half-duplex streams. Then, I began to convert character IO device drivers to streams. I wrote a tty editing module, based on Jim McKie's 'EUUG' tty driver. Some user-level programs were changed either to take advantage of the new structure, or to do work that the kernel once did. For example, `/etc/init` and `/etc/getty` push the tty module onto a terminal stream after opening it. Berkeley pseudo-terminals were discarded, and programs that used them were modified to use the appropriate streams mechanisms instead.

After each stage the new kernel was indeed smaller, even though some new functionality had been added (for instance, 'message' and 'connection' modules) [Pre85]. Certainly, one hopes for this result when replacing an accumulation of many special cases and conventions by something with a clean, general model. Much of the saving comes from eliminating device and line discipline 'read' and 'write' interface routines (the streams code does the work).

Sun added STREAMS to SunOS 4.0. In every case the comparable STREAMS versions have grown slightly larger, not smaller, than the components they replace. (The compatibility module `ttcompat` is smaller than the old tty driver, but it simply translates `ioctl` calls for the new tty module `ldterm`, which is larger.) The increase in the size of the tty modules on conversion to STREAMS is possibly because the new ones implement all the System V conventions. Unfortunately, some of those were artifacts of the old character IO system, and were overdue for replacement.

A good example is the `MIN/TIME` facility of the old terminal driver [Mic88]. It takes a full A4 page of typeset text to describe the effect of different combinations of `MIN` and `TIME` values, and the restrictions! One case provided buffering on a bursty asynchronous line (eg, for **uucp**): the driver waits until characters stop arriving frequently enough or a minimum number accumulates. Another allowed a process to wait until at least one character arrived or a timer expired. The first case could now be handled by pushing onto the stream a buffering module similar to *bufld*(4) in the Ninth Edition [Lab86], which can be applied to any stream, not just one with tty editing. (In fact, some programs that currently use `MIN/TIME` do not *want* tty editing.) The second looks like an application for `poll` or `select`. Instead, new mechanisms were added at the stream head to support the feature.

Streams also allow (indeed encourage) modules to be composed freely. Thus with streams, a Berkeley pseudo-terminal can be replaced by a pipe with message and tty modules installed appropriately. Neither module knows of the other's existence. (We implement simple text windows on our Sun 3/50s in this way; a more useful 'editing' shell window needs only the message module.) Of course, the particular application program must be revised accordingly, since the interfaces are radically different. In SunOS, the pseudo-terminals were instead 'converted' to streams, but their rather intricate nature required the use of an 'intra-stream' protocol between the pseudo-terminal parts and `ldterm`.

4.3. Conclusion

There is no point in trying blindly to reproduce or even to emulate all of the old mechanisms using the stream primitives, since the advantages of streams are quickly diminished, and the original point of adding them lost. 'Conversion' to streams provides a good opportunity to retire some of the older ideas that have seen their best days (when they were the only way to do something). For instance, the Berkeley pseudo-terminals could have been left in the system just as they were for compatibility, for a time, while people converted their programs to use the new streams facilities suitably. Eventually we could toss the old rubbish away, and be left with a clean, coherent system. Instead, we end up with new bugs in the old things, but old bugs in the new things!

5. RE-IMPLEMENTATION

A common theme of the paging and streams sections above is 're-implementation'. Many versions of UNIX were mainly ported and adapted to virtual memory machines, not thoroughly re-implemented. What is the distinction?

> By re-implementation, we mean re-programming and altering the structure of an existing system where necessary, so as to better implement the function of that system. We distinguish re-implementation from two other activities. These are the transportation of an existing system to new hardware, and the design of a new system. If we transport a system, we leave its function

> and its structure unaltered. Re-implementation is not merely a combination of transportation and re-design, but is a unique activity in its own right. *It is doing it again, but very well this time.*[She77] [my italics]

Re-implementation treats the old version of the system as a prototype of the desired system. Functions may be redistributed, added, or dropped. At no point, however, is the intent to provide a base for a completely different system: all the mechanisms are honed to implement precisely the key ideas of the original system.

For example, Andrew Hume's `mk`[Hum87] is a very good re-implementation of `make`: the underlying model and syntax of the two programs are obviously closely related, but in the new program the old ideas have been carefully refined, and it interacts more smoothly with the surrounding environment. In contrast, both Mach [Acc86] and Amoeba [Mul89] are new systems that can emulate the UNIX environment well enough to run UNIX programs, but the UNIX interface is not necessarily the best way to use them.

I think the essential ideas of UNIX are still worthwhile, but I also think it will take a good re-implementation to bring new life to them.

6. CONCLUSIONS

Although recent versions of UNIX have finally adopted some of the clever ideas from 'research' and elsewhere, this evidently has not resulted in a great reduction in system size or complexity. The new mechanisms have been modified (or mangled) to emulate the clumsy features they were intended to replace. It is important to keep a sense of proportion when deciding how much 'backwards compatibility' to maintain.

It is dangerous to place too much hope in any improvement coming from just following new fashions, if we lack insight into what really went wrong before. Without that insight, I suspect that rewriting UNIX in C++, for example, could easily become an excuse for increasing complexity (because by using C++ 'we can handle more complexity').

The system we use has been put together using a traditional UNIX approach: make a careful selection of mechanisms, and if possible, steal good ideas from elsewhere, but ensure they work together and fit the underlying system structure. The result is a more powerful system which is less greedy for memory (and my time). It still leaves room for Thompson's ''much-needed gap'' [Tho75].

Acknowledgements

Discussions with Martin Atkins about the state of UNIX and software in general have been particularly profitable, and he made very helpful comments on drafts of this paper. Martin, Peter Whysall and several others in the Department have been 'friendly users' of this software, made many good suggestions, and have given much encouragement. Dennis Ritchie very kindly sent a copy of the Ninth Edition manual.

References

Acc86. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, ''Mach: A New Kernel Foundation for UNIX Development'', *USENIX Association Conference Proceedings*, Atlanta, Georgia, pp. 93-113 (Summer 1986).

Akh87. P. Akhtar, ''A Replacement for Berkeley Memory Management'', *USENIX Association Conference Proceedings*, Phoenix, Arizona, pp. 69-80 (Summer 1987).

Bra89. A. Braunstein, M. Riley and J. Wilkes, ''Improving the efficiency of UNIX file buffer caches'', *Operating Systems Review* **23**(5), pp. 71-82 (1989).

Dun86. B. Duncanson, K. F. Storm and A. Facius, ''EUUG Technical Session Report: The Streams facility in UNIX System V rel. 3'', *European UNIX User Group Newsletter* **6**(1), p. 13, European UNIX System User Group Meeting Bella Center, Copenhagen - 10th - 13th September 1985 (Spring 1986).

Hum87. A. Hume, ''Mk: A Successor to Make'', *USENIX Association Conference Proceedings*, Phoenix, Arizona, pp. 445-458 (Summer 1987).

Kil84. T. J. Killian, ''Processes as Files'', *USENIX Association Conference Proceedings*, Salt Lake City, Utah, pp. 203-207 (Summer 1984).

Kle86. S. R. Kleiman, ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX'', *USENIX Association Conference Proceedings*, Atlanta, Georgia, pp. 238-247 (Summer 1986).

Lab86. AT&T Bell Laboratories, in *UNIX Programmer's Manual, Ninth Edition, Volume One*, ed. M. D. McIlroy, Murray Hill, New Jersey (September 1986).

Lef89. S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System,* Addison-Wesley, New York (1989).

Mic88. Sun Microsystems, pp. 1307 in *UNIX Interface Reference Manual, SunOS 4.0*, Mountain View, California (May 1988).

Mil84. R. Miller, ‘‘A Demand Paging Virtual Memory Manager for System V’’, *USENIX Association Conference Proceedings*, Salt Lake City, Utah, pp. 178-182 (Summer 1984).

Mul89. S. J. Mullender, ‘‘Amoeba - High-Performance Distributed Computing’’, *European UNIX Systems User Group Spring Conference*, Brussels, Belgium, pp. 17-26 (April 1989).

Pik83. R. Pike, ‘‘Graphics in Overlapping Bitmap Layers’’, *ACM Transactions on Graphics* **2**(2), pp. 135-160 (April 1983).

Pik84. R. Pike, ‘‘The Blit: A Multiplexed Graphics Terminal’’, *AT & T Bell Laboratories Technical Journal* **63**(8 part 2), pp. 1607-1632 (October 1984).

Pre85. D. L. Presotto and D. M. Ritchie, ‘‘Interprocess Communication in the 8th Edition UNIX System’’, *USENIX Association Conference Proceedings*, Portland, Oregon, pp. 309-316 (Summer 1985).

Pre86. D. L. Presotto, ‘‘The Eighth Edition Unix Connection Service’’, *European UNIX Systems User Group Autumn Conference*, Florence, Italy, pp. 1-9 (April 1986).

Rag89. Stephen Rago, ‘‘Out-Of-Band Communications in STREAMS’’, *Proc. 1989 Summer USENIX Conference*, San Francisco, California (June 1989).

Ree81. D. J. Rees, ‘‘The Structure of the EMAS 2900 Kernel’’, CSR-91-81, Unversity of Edinburgh, Dept of Computer Science (August 1981).

Rif86. A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah and K. Yueh, ‘‘RFS Architectural Overview’’, *European UNIX User Group Newsletter* **6**(2), pp. 13-23 (Summer 1986).

Rit84. D. M. Ritchie, ‘‘A Stream Input-Output System’’, *AT & T Bell Laboratories Technical Journal* **63**(8 part 2), pp. 1897-1910 (October 1984).

Rod86. R. Rodriguez, M. Koehler and R. Hyde, ‘‘The Generic File System’’, *USENIX Association Conference Proceedings*, Atlanta, Georgia, pp. 260-269 (Summer 1986).

Ros89. D. Rosenthal, ‘‘More Haste, Less Speed’’, *European UNIX Systems User Group Spring Conference*, Brussels, Belgium, pp. 123-130 (April 1989).

She74. N. A. Shelness, P. D. Stephens and H. Whitfield, ‘‘The Edinburgh Multi-Access System Scheduling and Allocation Procedures in theResident Supervisor’’, EMAS Report 4, University of Edinburgh, Department of Computer Science (April 1974).

She77. N. H. Shelness, D. J. Rees, P. D. Stephens and J. K. Yarwood, ‘‘An Experiment in Doing it Again, But Very Well This Time’’, CSR-18-77, Unversity of Edinburgh, Dept of Computer Science (December 1977).

Ste80. P. D. Stephens, J. K. Yarwood, D. J. Rees and N. H. Shelness, D. J. Rees and P. D. Stephens, ‘‘The Kernel of the EMAS 2900 Operating System’’, *SOFTWARE - Practice and Experience* **12**(7), pp. 655-668 (July 1982).

Str86. J. H. Straathof, A. K. Thareja and A. K. Agrawala, ‘‘UNIX Scheduling for Large Systems’’, *USENIX Association Conference Proceedings*, Denver, Colorado, pp. 111-139 (Winter 1986).

Tho75. K. Thompson, ‘‘The UNIX Command Language’’, pp. 375-384 in *Structured Programming—Infotech State of the Art Report,* Infotech International Ltd., Nicholson House, Maidenhead, Berkshire, England (March 1975).

Wei84. P. J. Weinberger, ‘‘The Version 8 Network File System (Abstract)’’, *USENIX Association Conference Proceedings*, Salt Lake City, Utah, p. 86 (Summer 1984).

Wil75. M. V. Wilkes, *Time-Sharing Computer Systems, Third Edition*, Macdonald and Jane’s, London (1975).