Performance Improvements and Functional Enhancements in 4.3BSD

M. Kirk McKusick
Mike Karels

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

Sam Leffler

Computer Division
Lucasfilm, Ltd.
PO Box 2009
San Rafael, California 94912

ABSTRACT

The 4.2 Berkeley Software Distribution of UNIX† for the VAX‡ provided many new facilities. This paper highlights the changes to 4.2BSD that appear in 4.3BSD. The changes to the system have consisted of improvements to the performance of the existing facilities, as well as enhancements to the current facilities. Performance improvements in the kernel include cacheing of path name translations, reductions in clock handling and scheduling overhead, and improved throughput of the network. Performance improvements in the libraries and utilities include replacement of linear searches of system databases with indexed lookup, merging of most network services into a single daemon, and conversion of system utilities to use the more efficient facilities available in 4.2BSD. Enhancements in the kernel include the addition of subnets and gateways, increases in many kernel limits, cleanup of the signal and autoconfiguration implementations, and support for windows and system logging. Functional extensions in the libraries and utilities include the addition of an Internet name server, new system management tools, and extensions to *dbx* to work with Pascal. The paper concludes with a brief discussion of changes made to the system to enhance security.

1. Introduction

The development effort for 4.2BSD concentrated on providing new facilities, and in getting them to work correctly. The limited development period left little time for tuning the completed system. The increased user feedback that came with the release of 4.2BSD and a growing body of experience with the system highlighted the performance shortcomings of 4.2BSD. With the release of 4.3BSD many of these problems have been addressed. The first part of this paper describes the performance improvements that have been made to the system; the second part describes functional enhancements that have been made; the final part discusses some of the

† UNIX is a trademark of Bell Laboratories.
‡ VAX, MASSBUS, UNIBUS, and DEC are trademarks of Digital Equipment Corporation.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Goverment.

security problems that have been addressed. Much of the work has been done in the machine independent parts of the system, hence these improvements could be applied to other variants of UNIX with equal success.

2. Performance Improvements

Techniques for measuring, benchmarking, and tuning 4.2BSD are described in [Leffler84]. The purpose of this paper is to report on more recent results of those efforts and to describe the implementations that we incorporated in the 4.3BSD release. The improvements fall into two major classes; changes to the kernel that are described in this section, and changes to the system libraries and utilities that are described in the following section.

2.1. Performance Improvements in the Kernel

Our goal has been to optimize system performance for our general timesharing environment. Since most sites running 4.2BSD have been forced to take advantage of declining memory costs rather than replace their existing machines with ones that are more powerful, we have chosen to optimize running time at the expense of memory. This tradeoff may need to be reconsidered for personal workstations that have smaller memories and higher latency disks. Decreases in the running time of the system may be unnoticeable because of higher paging rates incurred by a larger kernel. Where possible, we have allowed the size of caches to be controlled so that systems with limited memory may reduce them as appropriate.

Problem areas are identified by running long term profiling studies on general time sharing machines. Once a particular problem area is identified a "micro benchmark" is written to exercise a particular feature. When the kernel has been optimized for the benchmark, a long term profiling study is run to verify that the change works in normal use. Figure 1 describes several micro benchmarks that measure the speed of common operations handled by the kernel. Figure 2 shows the running time of the micro benchmarks on a 2 megabyte VAX 11/750 running 4.1BSD, 4.2BSD, and 4.3BSD. The major changes to the kernel are described in the following subsections.

2.1.1. Name Cacheing

Our initial profiling studies showed that more than one quarter of the time in the system was spent translating path names to inodes[1]. More detailed analysis showed two major sources of names to be translated. One major source of name translations came from requests to execute system utilities or access system databases such as the password file. The other major source of names to be translated came from programs making a linear scan of a directory.

Frequent requests for a small set of names are best handled with a cache of recent name translations. The system already maintained a cache of recently accessed inodes, so the initial name cache maintained a simple name-inode association that was used to check each component of a path name during name translations. We considered implementing the cache by tagging each inode with its most recently translated name, but eventually decided to have a separate data structure that kept names with pointers to the inode table. Tagging inodes has two drawbacks; many inodes such as those associated with login ports remain in the inode table for a long period of time, but are never looked up by name. Other inodes, such as those describing directories are looked up frequently by many different names (*e.g.* ".."). By keeping a separate table of names, the cache can truly reflect the most recently used names. An added benefit is that the table can be sized independently of the inode table, so that machines with small amounts of memory can reduce the size of the cache (or even eliminate it) without modifying the inode table structure.

Another issue to be considered is how the name cache should hold references to the inode table. Normally processes hold "hard references" by incrementing the reference count in the inode they reference. Since the system reuses only inodes with zero reference counts, a hard

[1] Inode is an abbreviation for "Index node". Each file on the system is described by an inode; the inode maintains access permissions, and an array of pointers to the disk blocks that hold the data associated with the file.

Test	Description
syscall	perform 100,000 *getpid* system calls
csw	perform 10,000 context switches using signals
signocsw	send 10,000 signals to yourself
pipeself4	send 10,000 4-byte messages to yourself
pipeself512	send 10,000 512-byte messages to yourself
pipediscard4	send 10,000 4-byte messages to child who discards
pipediscard512	send 10,000 512-byte messages to child who discards
pipeback4	exchange 10,000 4-byte messages with child
pipeback512	exchange 10,000 512-byte messages with child
forks0	fork-exit-wait 1,000 times
forks1k	sbrk(1024), fault page, fork-exit-wait 1,000 times
forks100k	sbrk(102400), fault pages, fork-exit-wait 1,000 times
vforks0	vfork-exit-wait 1,000 times
vforks1k	sbrk(1024), fault page, vfork-exit-wait 1,000 times
vforks100k	sbrk(102400), fault pages, vfork-exit-wait 1,000 times
execs0null	fork-exec "null job"-exit-wait 1,000 times
execs0null (1K env)	execs0null above, with 1K environment added
execs1knull	sbrk(1024), fault page, fork-exec "null job"-exit-wait 1,000 times
execs1knull (1K env)	execs1knull above, with 1K environment added
execs100knull	sbrk(102400), fault pages, fork-exec "null job"-exit-wait 1,000 times
vexecs0null	vfork-exec "null job"-exit-wait 1,000 times
vexecs1knull	sbrk(1024), fault page, vfork-exec "null job"-exit-wait 1,000 times
vexecs100knull	sbrk(102400), fault pages, vfork-exec "null job"-exit-wait 1,000 times
execs0big	fork-exec "big job"-exit-wait 1,000 times
execs1kbig	sbrk(1024), fault page, fork-exec "big job"-exit-wait 1,000 times
execs100kbig	sbrk(102400), fault pages, fork-exec "big job"-exit-wait 1,000 times
vexecs0big	vfork-exec "big job"-exit-wait 1,000 times
vexecs1kbig	sbrk(1024), fault pages, vfork-exec "big job"-exit-wait 1,000 times
vexecs100kbig	sbrk(102400), fault pages, vfork-exec "big job"-exit-wait 1,000 times

Figure 1. Benchmark programs.

reference insures that the inode pointer will remain valid. However, if the name cache holds hard references, it is limited to some fraction of the size of the inode table, since some inodes must be left free for new files. It also makes it impossible for other parts of the kernel to verify sole use of a device or file. These reasons made it impractical to use hard references without affecting the behavior of the inode cacheing scheme. Thus, we chose instead to keep "soft references" protected by a *capability* – a 32-bit number guaranteed to be unique[2]. When an entry is made in the name cache, the capability of its inode is copied to the name cache entry. When an inode is reused it is issued a new capability. When a name cache hit occurs, the capability of the name cache entry is compared with the capability of the inode that it references. If the capabilities do not match, the name cache entry is invalid. Since the name cache holds only soft references, it may be sized independent of the size of the inode table. A final benefit of using capabilities is that all cached names for an inode may be invalidated without searching through the entire cache; instead all you need to do is assign a new capability to the inode.

The name cache is able to resolve more than 70% of the names passed to the system. However, it does not provide any help for programs that sequentially scan a directory. These names are not usually in the cache and, once operated on, they are unlikely to be accessed again. To improve performance for processes doing directory scans, the system keeps track of the directory

[2] When all the numbers have been exhausted, all outstanding capabilities are purged and numbering starts over from scratch. Purging is possible as all capabilities are easily found in kernel memory.

Test	Real			User			System		
	4.1	4.2	4.3	4.1	4.2	4.3	4.1	4.2	4.3
syscall	28.0	29.0	23.0	4.5	5.3	3.5	23.9	23.7	20.4
csw	45.0	60.0	45.0	3.5	4.3	3.3	19.5	25.4	19.0
signocsw	16.5	23.0	16.0	1.9	3.0	1.1	14.6	20.1	15.2
pipeself4	21.5	29.0	26.0	1.1	1.1	0.8	20.1	28.0	25.6
pipeself512	47.5	59.0	55.0	1.2	1.2	1.0	46.1	58.3	54.2
pipediscard4	32.0	42.0	36.0	3.2	3.7	3.0	15.5	18.8	15.6
pipediscard512	61.0	76.0	69.0	3.1	2.1	2.0	29.7	36.4	33.2
pipeback4	57.0	75.0	66.0	2.9	3.2	3.3	25.1	34.2	29.7
pipeback512	110.0	138.0	125.0	3.1	3.4	2.2	52.2	65.7	57.7
forks0	37.5	41.0	22.0	0.5	0.3	0.3	34.5	37.6	21.5
forks1k	40.0	43.0	22.0	0.4	0.3	0.3	36.0	38.8	21.6
forks100k	217.5	223.0	176.0	0.7	0.6	0.4	214.3	218.4	175.2
vforks0	34.5	37.0	22.0	0.5	0.6	0.5	27.3	28.5	17.9
vforks1k	35.0	37.0	22.0	0.6	0.8	0.5	27.2	28.6	17.9
vforks100k	35.0	37.0	22.0	0.6	0.8	0.6	27.6	28.9	17.9
execs0null	97.5	92.0	66.0	3.8	2.4	0.6	68.7	82.5	48.6
execs0null (1K env)	197.0	229.0	75.0	4.1	2.6	0.9	167.8	212.3	62.6
execs1knull	99.0	100.0	66.0	4.1	1.9	0.6	70.5	86.8	48.7
execs1knull (1K env)	199.0	230.0	75.0	4.2	2.6	0.7	170.4	214.9	62.7
execs100knull	283.5	278.0	216.0	4.8	2.8	1.1	251.9	269.3	202.0
vexecs0null	100.0	92.0	66.0	5.1	2.7	1.1	63.7	76.8	45.1
vexecs1knull	100.0	91.0	66.0	5.2	2.8	1.1	63.2	77.1	45.1
vexecs100knull	100.0	92.0	66.0	5.1	3.0	1.1	64.0	77.7	45.6
execs0big	129.0	201.0	101.0	4.0	3.0	1.0	102.6	153.5	92.7
execs1kbig	130.0	202.0	101.0	3.7	3.0	1.0	104.7	155.5	93.0
execs100kbig	318.0	385.0	263.0	4.8	3.1	1.1	286.6	339.1	247.9
vexecs0big	128.0	200.0	101.0	4.6	3.5	1.6	98.5	149.6	90.4
vexecs1kbig	125.0	200.0	101.0	4.7	3.5	1.3	98.9	149.3	88.6
vexecs100kbig	126.0	200.0	101.0	4.2	3.4	1.3	99.5	151.0	89.0

Table 2. Benchmark results (all times in seconds).

offset of the last component of the most recently translated path name for each process. If the next name the process requests is in the same directory, the search is started from the point that the previous name was found (instead of from the beginning of the directory). This changes the previous $O(n^2)$ algorithm to a (potentially) $O(n)$ algorithm.

On our general time sharing systems we find that during the twelve hour period from 8AM to 8PM the system does 500,000 to 1,000,000 name translations. The name cache has a hit rate of 70%-80%; the directory offset cache gets a hit rate of 5%-15%. The combined hit rate of the two caches almost always adds up to 85%. With the addition of the two caches, the percentage of system time devoted to name translation has dropped from 25% to less than 10%.

2.1.2. Intelligent Auto Siloing

Most VAX terminal input hardware can run in two modes: it can either generate an interrupt each time a character is received, or collect characters in a silo that the system then periodically drains. To provide quick response for interactive input and flow control, a silo must be checked 30 to 50 times per second. Ascii terminals normally exhibit an input rate of less than 30 characters per second so, they are most efficiently handled with interrupt per character mode. When input is being generated by another machine, however, the input rate is usually more than 50 characters per second so, it is more efficient to use a device's silo input mode. Since a given dialup port may switch between uucp logins and user logins, it is impossible to statically select the most efficient input mode to use. Thus, the system monitors the character input rate; input is handled on an interrupt at a time basis during periods of low input rates, and with the hardware silos during periods of high input rates.

2.1.3. Process Table Management

As systems have grown larger, the size of the process table has grown far past 200 entries. With large tables, linear searches must be eliminated from any frequently used facility. The kernel process table is now multi-threaded to allow selective searching of active and zombie processes. A third list threads unused process table slots. Free slots can be obtained in constant time by taking one from the front of the free list. The number of processes used by a given user may be computed by scanning only the active list. Since the 4.2BSD release, the kernel maintained linked lists of the descendents of each process. This linkage is now exploited when dealing with process exit; parents seeking the exit status of children now avoid linear search of the process table, but examine only their direct descendents. In addition, the previous algorithm for finding all descendents of an exiting process used multiple linear scans of the process table. This has been changed to follow the links between child process and siblings.

When forking a new process, the system must assign it a unique process identifier. The system previously scanned the entire process table each time it created a new process to locate an identifier that was not already in use. Now, to avoid scanning the process table for each new process, the system computes a range of unused identifiers that can be directly assigned. Only when the set of identifiers is exhausted is another process table scan required.

2.1.4. Scheduling

Previously the scheduler scanned the entire process table once per second to recompute process priorities. Processes that had run for their entire time slice had their priority lowered. Processes that had not used their time slice, or that had been sleeping for the past second had their priority raised. On systems running many processes, the scheduler represented nearly 20% of the system time. To reduce this overhead, the scheduler has been changed to consider only runnable processes when recomputing priorities. To insure that processes sleeping for more than a second still get their appropriate priority boost, their priority is recomputed when they are placed back on the run queue. Since the set of runnable process is typically only a small fraction of the total number of processes on the system, the cost of invoking the scheduler drops proportionally.

2.1.5. Clock Handling

The hardware clock interrupts the processor 100 times per second at high priority. As most of the clock-based events need not be done at high priority, the system schedules a lower priority software interrupt to do the less time-critical events such as cpu scheduling and timeout processing. Often there are no such events, and the software interrupt handler finds nothing to do and returns. The high priority event now checks to see if there are low priority events to process; if there is nothing to do, the software interrupt is not requested. Often, the high priority interrupt occurs during a period when the machine had been running at low priority. Rather than posting a software interrupt that would occur as soon as it returns, the hardware clock interrupt handler simply lowers the processor priority and calls the software clock routines directly. Between these two optimizations, nearly 80 of the 100 software interrupts per second can be eliminated.

2.1.6. File System

The file system uses a large block size, typically 4096 or 8192 bytes. To allow small files to be stored efficiently, the large blocks can be broken into smaller fragments, typically multiples of 1024 bytes. To minimize the number of full-sized blocks that must be broken into fragments, the file system uses a best fit strategy. Programs that slowly grow files using write of 1024 bytes or less can force the file system to copy the data to successively larger and larger fragments until it finally grows to a full sized block. The file system still uses a best fit strategy the first time a fragment is written. However, the first time that the file system is forced to copy a growing fragment it places it at the beginning of a full sized block. Continued growth can be accommodated without further copying by using up the rest of the block. If the file ceases to grow, the rest of the block is still available for holding other fragments.

When creating a new file name, the entire directory in which it will reside must be scanned to insure that the name does not already exist. For large directories, this scan is time consuming. Because there was no provision for shortening directories, a directory that is once over-filled will increase the cost of file creation even after the over-filling is corrected. Thus, for example, a congested uucp connection can leave a legacy long after it is cleared up. To alleviate the problem, the system now deletes empty blocks that it finds at the end of a directory while doing a complete scan to create a new name.

2.1.7. Network

The default amount of buffer space allocated for stream sockets (including pipes) has been increased to 4096 bytes. Stream sockets and pipes now return their buffer sizes in the block size field of the stat structure. This information allows the standard I/O library to use more optimal buffering. Unix domain stream sockets also return a dummy device and inode number in the stat structure to increase compatibility with other pipe implementations. The TCP maximum segment size is calculated according to the destination and interface in use; non-local connections use a more conservative size for long-haul networks.

On multiply-homed hosts, the local address bound by TCP now always corresponds to the interface that will be used in transmitting data packets for the connection. Several bugs in the calculation of round trip timing have corrected. TCP now switches to an alternate gateway when an existing route fails, or when an ICMP redirect message is received. ICMP source quench messages are used to throttle the transmission rate of TCP streams by temporarily creating an artificially small send window, and retransmissions send only a single packet rather than resending all queued data. A send policy has been implemented that decreases the number of small packets outstanding for network terminal traffic [Nagle84], providing additional reduction of network congestion. The overhead of packet routing has been decreased by changes in the routing code and by cacheing the most recently used route for each datagram socket.

2.1.8. Exec

When *exec*-ing a new process, the kernel creates the new program's argument list by copying the arguments and environment from the parent process's address space into the system, then back out again onto the stack of the newly created process. These two copy operations were done one byte at a time, but are now done a string at a time. This optimization reduced the time to process an argument list by a factor of ten; the average time to do an *exec* call decreased by 25%.

2.1.9. Context Switching

The kernel used to post a software event when it wanted to force a process to be rescheduled. Often the process would be rescheduled for other reasons before exiting the kernel, delaying the event trap. At some later time the process would again be selected to run and would complete its pending system call, finally causing the event to take place. The event would cause the scheduler to be invoked a second time selecting the same process to run. The fix to this problem is to cancel any software reschedule events when saving a process context. This change doubles the speed with which processes can synchronize using pipes or signals.

2.1.10. Setjmp/Longjmp

The kernel routine *setjmp*, that saves the current system context in preparation for a non-local goto used to save many more registers than necessary under most circumstances. By trimming its operation to save only the minimum state required, the overhead for system calls decreased by an average of 13%.

2.1.11. Compensating for Lack of Compiler Technology

The current compilers available for C do not do any significant optimization. Good optimizing compilers are unlikely to be built; the C language is not well suited to optimization because of its rampant use of unbound pointers. Thus, many classical optimizations such as common subexpression analysis and selection of register variables must be done by hand using "exterior" knowledge of when such optimizations are safe.

Another optimization usually done by optimizing compilers is inline expansion of small or frequently used routines. In past Berkeley systems this has been done by using *sed* to run over the assembly language and replace calls to small routines with the code for the body of the routine, often a single VAX instruction. While this optimization eliminated the cost of the subroutine call and return, it did not eliminate the pushing and popping of several arguments to the routine. The *sed* script has been replaced by a more intelligent expander, *inline*, that merges the pushes and pops into moves to registers. For example, if the C code

```
if (scanc(map[i], 1, 47, i - 63))
```

is compiled into assembly language it generates the code shown in the left hand column of Figure 3. The *sed* inline expander changes this code to that shown in the middle column. The newer optimizer eliminates most of the stack operations to generate the code shown in the right hand column.

cc		sed		inline	
subl3	$64,_i,-(sp)	subl3	$64,_i,-(sp)	subl3	$64,_i,r5
pushl	$47	pushl	$47	movl	$47,r4
pushl	$1	pushl	$1	pushl	$1
mull2	$16,_i,r3	mull2	$16,_i,r3	mull2	$16,_i,r3
pushl	-56(fp)[r3]	pushl	-56(fp)[r3]	movl	-56(fp)[r3],r2
calls	$4,_scanc	movl	(sp)+,r5	movl	(sp)+,r3
tstl	r0	movl	(sp)+,r4	scanc	r2,(r3),(r4),r5
jeql	L7	movl	(sp)+,r3	tstl	r0
		movl	(sp)+,r2	jeql	L7
		scanc	r2,(r3),(r4),r5		
		tstl	r0		
		jeql	L7		

Figure 3. Inline code expansion.

Another optimization involved reevaluating existing data structures in the context of the current system. For example, disk buffer hashing was implemented when the system typically had thirty to fifty buffers. Most systems today have 200 to 1000 buffers. Consequently, most of the hash chains contained ten to a hundred buffers each! The running time of the low level buffer management primitives was dramatically improved simply by enlarging the size of the hash table.

2.2. Improvements to Libraries and Utilities

Intuitively, changes to the kernel would seem to have the greatest payoff since they affect all programs that run on the system. However, the kernel has been tuned many times before, so the opportunity for significant improvement is small. By contrast, many of the libraries and utilities have never been tuned. For example, we have found utilities that spent 90% of their running time doing single character read system calls. Changing the utility to use the standard I/O library cut the running time by a factor of five! Thus, while most of our time has been spent tuning the kernel, more than half of the speedups are because of improvements in other parts of the system. Some of the more dramatic changes are described in the following subsections.

2.2.1. Hashed Databases

UNIX provides a set of database management routines, *dbm*, that can be used to speed lookups in large data files with an external hashed index file. The original version of dbm was designed to work with only one database at a time. These routines were generalized to handle multiple database files, enabling them to be used in rewrites of the password and host file lookup routines. The new routines used to access the password file significantly improve the running time of many important programs such as the mail subsystem, the C-shell (in doing tilde expansion), *ls -l*, etc.

2.2.2. Buffered I/O

The standard error file (**stderr**) is now buffered to do only a single write for each call into the standard I/O library routines (*e.g.* fprintf). This buffering still allows partial line writes, but without requiring character at a time output that can congest a network. Several important utilities that did not use the standard I/O library and were buffering I/O using the old optimal buffer size, 1Kbytes; the programs were changed to buffer I/O according to the optimal file system blocksize. These include the editor, the assembler, loader, remote file copy, the text formatting programs, and the C compiler.

2.2.3. Mail System

The mail system previously used *link* and *unlink* in implementing file locking primitives. Because these operations usually modify the contents of directories they require synchronous disk operations and cannot take advantage of the name cache maintained by the system. Unlink requires that the entry be found in the directory so that it can be removed; link requires that the directory be scanned to insure that the name does not already exist. By contrast the advisory locking facility in 4.2BSD is efficient because it is all done with in-memory tables. Thus, the mail system was modified to use the file locking primitives; it also benefited from the database hashing and extensive profiling and tuning of *sendmail.*

2.2.4. Network Servers

With the introduction of the network facilities in 4.2BSD, a myriad of services became available, each of which required its own daemon process. Many of these daemons were rarely if ever used, yet they lay asleep in the process table consuming system resources and generally slowing down response. Most of the daemons were eliminated by merging them into a single "Internet daemon" that listens on all the service ports and only forks a server process when a request for their service arrives. This allowed as many as twenty processes to be eliminated.

2.2.5. The C Run-time Library

Several people have found poorly tuned code in frequently used routines in the C library [Lankford84]. In particular the running time of the string routines can be cut in half by rewriting them using the VAX string instructions. The memory allocation routines have been tuned to waste less memory for memory allocations with sizes that are a power of two. Certain library routines that did file input in one-character reads have been corrected. Other library routines including *fread* and *fwrite* have been rewritten for efficiency.

2.2.6. Csh

The C-shell was converted to run on 4.2BSD by writing a set of routines to simulate the old jobs library. While this provided a functioning C-shell, it was grossly inefficient, generating up to twenty system calls per prompt. The C-shell has been modified to use the new signal facilities directly, cutting the number of system calls per prompt in half. Additional tuning was done with the help of profiling to cut the cost of frequently used facilities.

3. Functional Extensions

Some of the facilities introduced in 4.2BSD were not completely implemented. An important part of the effort that went into 4.3BSD was to clean up and unify both new and old facilities.

3.1. Kernel Extensions

A significant effort went into improving the networking part of the kernel. The work consisted of fixing bugs, tuning the algorithms, and revamping the lowest levels of the system to better handle heterogeneous network topologies.

3.1.1. Subnets, Broadcasts and Gateways

To allow sites to expand their network in an autonomous and orderly fashion, subnetworks have been introduced in 4.3BSD [GADS85]. This facility allows sites to subdivide their local Internet address space into multiple subnetwork address spaces that are visible only by hosts at that site. To off-site hosts machines on a site's subnetworks appear to reside on a single network. The routing daemon has been reworked to provide routing support in this type of environment.

The default Internet broadcast address is now specified with a host part of all one's, rather than all zero's. The broadcast address may be set at boot time on a per-interface basis.

3.1.2. Interface Addressing

The organization of network interfaces has been reworked to more cleanly support multiple network protocols. Network interfaces no longer contain a host's address on that network; instead each interface contains a pointer to a list of addresses assigned to that interface. This permits a single interface to support, for example, Internet protocols at the same time as XNS protocols.

The Address Resolution Protocol (ARP) support for 10 megabyte/second Ethernet† has been made more flexible by allowing hosts to act as an "clearing house" for hosts that do not support ARP. In addition, system managers have more control over the contents of the ARP translation cache and may interactively interrogate and modify the cache's contents.

3.1.3. User Control of Network Buffering

Although the system allocates reasonable default amounts of buffering for most connections, certain operations such as file system dumps to remote machines benefit from significant increases in buffering [Walsh84]. The *setsockopt* system call has been extended to allow such requests. In addition, *getsockopt* and *setsockopt*, are now interfaced to the protocol level allowing protocol-specific options to be manipulated by the user.

3.1.4. Number of File Descriptors

To allow full use of the many descriptor based services available, the previous hard limit of 30 open files per process has been relaxed. The changes entailed generalizing *select* to handle arrays of 32-bit words, removing the dependency on file descriptors from the page table entries, and limiting most of the linear scans of a process's file table. The default per-process descriptor limit was raised from 20 to 64, though there are no longer any hard upper limits on the number of file descriptors.

3.1.5. Kernel Limits

Many internal kernel configuration limits have been increased by suitable modifications to data structures. The limit on physical memory has been changed from 8 megabyte to 64 megabyte, and the limit of 15 mounted file systems has been changed to 255. The maximum file

† Ethernet is a trademark of Xerox.

system size has been increased to 8 gigabyte, number of processes to 65536, and per process size to 64 megabyte of data and 64 megabyte of stack. Note that these are upper bounds, the default limits for these quantities are tuned for systems with 4-8 megabyte of physical memory.

3.1.6. Memory Management

The global clock page replacement algorithm used to have a single hand that was used both to mark and to reclaim memory. The first time that it encountered a page it would clear its reference bit. If the reference bit was still clear on its next pass across the page, it would reclaim the page. The use of a single hand does not work well with large physical memories as the time to complete a single revolution of the hand can take up to a minute or more. By the time the hand gets around to the marked pages, the information is usually no longer pertinent. During periods of sudden shortages, the page daemon will not be able to find any reclaimable pages until it has completed a full revolution. To alleviate this problem, the clock hand has been split into two separate hands. The front hand clears the reference bits, the back hand follows a constant number of pages behind reclaiming pages that still have cleared reference bits. While the code has been written to allow the distance between the hands to be varied, we have not found any algorithms suitable for determining how to dynamically adjust this distance.

The configuration of the virtual memory system used to require a significant understanding of its operation to carry out simple tasks such as increasing the maximum process size. This process has been significantly improved so that the most common configuration parameters, such as the virtual memory sizes, can be specified using a single option in the configuration file[3]. Standard configurations support data and stack segments of 17, 33 and 64 megabytes.

3.1.7. Signals

The 4.2BSD signal implementation would push several words onto the normal run-time stack before switching to an alternate signal stack. The 4.3BSD implementation has been corrected so that the entire signal handler's state is now pushed onto the signal stack. Another limitation in the original signal implementation was that it used an undocumented system call to return from signals. Users could not write their own return from exceptions; 4.3BSD formally specifies the *sigreturn* system call.

Many existing programs depend on interrupted system calls. The restartable system call semantics of 4.2BSD signals caused many of these programs to break. To simplify porting of programs from inferior versions of UNIX the *sigvec* system call has been extended so that programmers may specify that system calls are not to be restarted after particular signals.

3.1.8. System Logging

A system logging facility has been added that sends kernel messages to the syslog daemon for logging in /usr/adm/messages and possibly for printing on the system console. The revised scheme for logging messages eliminates the time lag in updating the messages file, unifies the format of kernel messages, provides a finer granularity of control over the messages that get printed on the console, and eliminates the degradation in response during the printing of low-priority kernel messages. Recoverable system errors and common resource limitations are logged using this facility. Most system utilities such as init and login, have been modified to log errors to syslog rather than writing directly on the console.

3.1.9. Windows

The tty structure has been augmented to hold information about the size of an associated window or terminal. These sizes can be obtained by programs such as editors that want to know the size of the screen they are manipulating. When these sizes are changed, a new signal,

[3] The rumor that former Berkeley gurus were making significant incomes by consulting on how to increase the maximum process size is completely untrue.

SIGWINCH, is sent the current process group. The editors have been modified to catch this signal and reshape their view of the world, and the remote login program and server now cooperate to propagate window sizes and window size changes across a network. Other programs and libraries such as curses that need the width or height of the screen have been modified to use this facility as well.

3.1.10. Configuration of UNIBUS Devices

The UNIBUS configuration routines have been extended to allow auto-configuration of dedicated UNIBUS memory held by devices. The new routines simplify the configuration of memory-mapped devices and correct problems occurring on reset of the UNIBUS.

3.1.11. Disk Recovery from Errors

The MASSBUS disk driver's error recovery routines have been fixed to retry before correcting ECC errors, support ECC on bad-sector replacements, and correctly attempt retries after earlier corrective actions in the same transfer. The error messages are more accurate.

3.2. Functional Extensions to Libraries and Utilities

Most of the changes to the utilities and libraries have been to allow them to handle a more general set of problems, or to handle the same set of problems more quickly.

3.2.1. Name Server

In 4.2BSD the name resolution routines (*gethostbyname*, *getservbyname*, etc.) were implemented by a set of database files maintained on the local machine. Inconsistencies or obsolescence in these files resulted in inaccessibility of hosts or services. In 4.3BSD these files may be replaced by a network name server that can insure a consistent view of the name space in a multimachine environment. This nameserver operates in accordance with Internet standards for service on the ARPANET [Mockapetris83].

3.2.2. System Management

A new utility, *rdist*, has been provided to assist system managers in keeping all their machines up to date with a consistent set of sources and binaries. A master set of sources may reside on a single central machine, or be distributed at (known) locations throughout the environment. New versions of *getty*, *init*, and *login* merge the functions of several files into a single place, and allow more flexibility in the startup of processes such as window managers.

A new utility keeps the time on a group of cooperating machines synchronized to within 30 milliseconds. It does its corrections using a new system call that changes the rate of time advance without stopping or reversing the system clock. It normally selects one machine to act as a master. If the master dies or is partitioned, a new master is elected. Other machines may participate in a purely slave role.

3.2.3. Routing

Many bugs in the routing daemon have been fixed; it is considerably more robust, and now understands how to properly deal with subnets and point-to-point networks. Its operation has been made more efficient by tuning with the use of execution profiles, along with inline expansion of common operations using the kernel's *inline* optimizer.

3.2.4. Compilers

The symbolic debugger *dbx* has had many new features added, and all the known bugs fixed. In addition *dbx* has been extended to work with the Pascal compiler. The fortran compiler *f77* has had numerous bugs fixed. The C compiler has been modified so that it can, optionally, generate single precision floating point instructions when operating on single precision variables.

4. Security Tightening

Since we do not wish to encourage rampant system cracking, we describe only briefly the changes made to enhance security.

4.1. Generic Kernel

Several loopholes in the process tracing facility have been corrected. Programs being traced may not be executed; executing programs may not be traced. Programs may not provide input to terminals to which they do not have read permission. The handling of process groups has been tightened to eliminate some problems. When a program attempts to change its process group, the system checks to see if the process with the pid of the process group was started by the same user. If it exists and was started by a different user the process group number change is denied.

4.2. Security Problems in Utilities

Setuid utilities no longer use the *popen* or *system* library routines. Access to the kernel's data structures through the kmem device is now restricted to programs that are set group id "kmem". Thus many programs that used to run with root privileges no longer need to do so. Access to disk devices is now controlled by an "operator" group id; this permission allows operators to function without being the super-user. Only users in group wheel can do "su root"; this restriction allows administrators to define a super-user access list. Numerous holes have been closed in the shell to prevent users from gaining privileges from set user id shell scripts, although use of such scripts is still highly discouraged on systems that are concerned about security.

5. References

[GADS85] GADS (Gateway Algorithms and Data Structures Task Force), "Toward an Internet Standard for Subnetting," RFC-940, Network Information Center, SRI International, April 1985.

[Lankford84] Jeffrey Lankford, "UNIX System V and 4BSD Performance," *Proceedings of the Salt Lake City Usenix Conference*, pp 228-236, June 1984.

[Leffler84] Sam Leffler, Mike Karels, and M. Kirk McKusick, "Measuring and Improving the Performance of 4.2BSD," *Proceedings of the Salt Lake City Usenix Conference*, pp 237-252, June 1984.

[Mockapetris83] Paul Mockapetris, "Domain Names – Implementation and Schedule," Network Information Center, SRI International, RFC-883, November 1983.

[Mogul84] Jeffrey Mogul, "Broadcasting Internet Datagrams," RFC-919, Network Information Center, SRI International, October 1984.

[Nagle84] John Nagle, "Congestion Control in IP/TCP Internetworks," RFC-896, Network Information Center, SRI International, January 1984.

[Walsh84] Robert Walsh and Robert Gurwitz, "Converting BBN TCP/IP to 4.2BSD," *Proceedings of the Salt Lake City Usenix Conference*, pp 52-61, June 1984.