Measuring and Improving the Performance of Berkeley UNIX*

November 30, 1985

Marshall Kirk McKusick,
Samuel J. Leffler†,
Michael J. Karels,
Luis Felipe Cabrera‡

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

ABSTRACT

The 4.2 Berkeley Software Distribution of UNIX† for the VAX‡ had several problems that could severely affect the overall performance of the system. These problems were identified with kernel profiling and system tracing during day to day use. Once potential problem areas had been identified benchmark programs were devised to highlight the bottlenecks. These benchmarks verified that the problems existed and provided a metric against which to validate proposed solutions. This paper examines the performance problems encountered and describes modifications that have been made to the system since the initial distribution.

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 subsystem. 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. All of these enhancements are present in Berkeley UNIX 4.3BSD.

---

* UNIX is a trademark of AT&T Bell Laboratories.

† Samuel J. Leffler is currently employed by: Lucasfilm Ltd., PO Box 2009, San Rafael, CA 94912

‡ Luis Felipe Cabrera is currently employed by: Computer Science Department, IBM Research Laboratory, 5600 Cottle Road, San Jose, California 95193.

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under ARPA Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

† UNIX is a Trademark of Bell Laboratories.

‡ VAX, MASSBUS, UNIBUS, and DEC are trademarks of Digital Equipment Corporation.

CR Categories and Subject Descriptors: D.4.3 **[Operating Systems]**: File Systems Management − *file organization, directory structures, access methods*; D.4.2 **[Operating Systems]**: Storage Management − *allocation/deallocation strategies, secondary storage devices*; D.4.8 **[Operating Systems]**: Performance − *measurements, operational analysis*; H.3.2 **[Information Systems]**: Information Storage − *file organization*

Additional Keywords and Phrases: Berkeley UNIX, file system organization, file system performance, file system design, application program interface.

General Terms: file system, measurement, performance.

TABLE OF CONTENTS

1. Introduction

The Berkeley Software Distributions of UNIX for the VAX have added many new capabilities that were previously unavailable under UNIX. The development effort for 4.2BSD concentrated on providing new facilities, and in getting them to work correctly. Many new data structures were added to the system to support these new capabilities. In addition, many of the existing data structures and algorithms were put to new uses or their old functions placed under increased demand. The effect of these changes was that mechanisms that were well tuned under 4.1BSD no longer provided adequate performance for 4.2BSD. 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.

This paper details the work that we have done since the release of 4.2BSD to measure the performance of the system, detect the bottlenecks, and find solutions to remedy them. Most of our tuning has been in the context of the real timesharing systems in our environment. Rather than using simulated workloads, we have sought to analyze our tuning efforts under realistic conditions. 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. All of the changes made have been included in 4.3BSD.

Section 2 of the paper describes the tools and techniques available to us for measuring system performance. In Section 3 we present the results of using these tools, while Section 4 has the performance improvements that have been made to the system based on our measurements. Section 5 highlights the functional enhancements that have been made to Berkeley UNIX 4.2BSD. Section 6 discusses some of the security problems that have been addressed.

2. Observation techniques

There are many tools available for monitoring the performance of the system. Those that we found most useful are described below.

2.1. System maintenance tools

Several standard maintenance programs are invaluable in observing the basic actions of the system. The *vmstat*(1) program is designed to be an aid to monitoring systemwide activity. Together with the *ps*(1) command (as in ‘‘ps av’’), it can be used to investigate systemwide virtual memory activity. By running *vmstat* when the system is active you can judge the system activity in several dimensions: job distribution, virtual memory load, paging and swapping activity, disk and cpu utilization. Ideally, to have a balanced system in activity, there should be few blocked (b) jobs, there should be little paging or swapping activity, there should be available bandwidth on the disk devices (most single arms peak out at 25-35 tps in practice), and the user cpu utilization (us) should be high (above 50%).

If the system is busy, then the count of active jobs may be large, and several of these jobs may often be blocked (b). If the virtual memory is active, then the paging demon will be running (sr will be non-zero). It is healthy for the paging demon to free pages when the virtual memory gets active; it is triggered by the amount of free memory dropping below a threshold and increases its pace as free memory goes to zero.

If you run *vmstat* when the system is busy (a ‘‘vmstat 5’’ gives all the numbers computed by the system), you can find imbalances by noting abnormal job distributions. If many processes are blocked (b), then the disk subsystem is overloaded or imbalanced. If you have several non-dma devices or open teletype lines that are ‘‘ringing’’, or user programs that are doing high-speed non-buffered input/output, then the system time may go high (60-80% or higher). It is often possible to pin down the cause of high system time by looking to see if there is excessive context switching (cs), interrupt activity (in) or system call activity (sy). Long term measurements on one of our large machines show an average of 60 context switches and interrupts per second and an average of 90 system calls per second.

If the system is heavily loaded, or if you have little memory for your load (1 megabyte is little in our environment), then the system may be forced to swap. This is likely to be accompanied by a noticeable reduction in the system responsiveness and long pauses when interactive jobs such as editors swap out.

A second important program is *iostat*(1). *Iostat* iteratively reports the number of characters read and written to terminals, and, for each disk, the number of transfers per second, kilobytes transferred per second, and the milliseconds per average seek. It also gives the percentage of time the system has spent in user mode, in user mode running low priority (niced) processes, in system mode, and idling.

To compute this information, for each disk, seeks and data transfer completions and the number of words transferred are counted; for terminals collectively, the number of input and output characters are counted. Also, every 100 ms, the state of each disk is examined and a tally is made if the disk is active. From these numbers and the transfer rates of the devices it is possible to determine average seek times for each device.

When filesystems are poorly placed on the available disks, figures reported by *iostat* can be used to pinpoint bottlenecks. Under heavy system load, disk traffic should be spread out among the drives with higher traffic expected to the devices where the root, swap, and /tmp filesystems are located. When multiple disk drives are attached to the same controller, the system will attempt to overlap seek operations with I/O transfers. When seeks are performed, *iostat* will show non-zero average seek times. Most modern disk drives should exhibit an average seek time of 25-35 ms.

Terminal traffic reported by *iostat* should be heavily output oriented unless terminal lines are being used for data transfer by programs such as *uucp*. Input and output rates are system specific. Screen editors such as *vi* and *emacs* tend to exhibit output/input ratios of anywhere from 5/1 to 8/1. On one of our largest systems, 88 terminal lines plus 32 pseudo terminals, we observed an average of 180 characters/second input and 450 characters/second output over 4 days of operation.

2.2. Kernel profiling

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

Once a profiling system has been booted statistic gathering is handled by *kgmon* (8). *Kgmon* allows profiling to be started and stopped and the internal state of the profiling buffers to be dumped. *Kgmon* can also be used to reset the state of the internal buffers to allow multiple experiments to be run without rebooting the machine.

The profiling data is processed with *gprof* (1) to obtain information regarding the system's operation. Profiled systems maintain histograms of the kernel program counter, the number of invocations of each routine, and a dynamic call graph of the executing system. The postprocessing propagates the time spent in each routine along the arcs of the call graph. *Gprof* then generates a listing for each routine in the kernel, sorted according to the time it uses including the time of its call graph descendents. Below each routine entry is shown its (direct) call graph children, and how their times are propagated to this routine. A similar display above the routine shows how this routine's time and the time of its descendents is propagated to its (direct) call graph parents.

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

2.3. Kernel tracing

The kernel can be configured to trace certain operations by specifying ''options TRACE'' in the configuration file. This forces the inclusion of code that records the occurrence of events in *trace records* in a circular buffer in kernel memory. Events may be enabled/disabled selectively while the system is operating. Each trace record contains a time stamp (taken from the VAX hardware time of day clock register), an event identifier, and additional information that is interpreted according to the event type. Buffer cache operations, such as initiating a read, include the disk drive, block number, and transfer size in the trace record. Virtual memory operations, such as a pagein completing, include the virtual address and process id in the trace record. The circular buffer is normally configured to hold 256 16-byte trace records.

Several user programs were written to sample and interpret the tracing information. One program runs in the background and periodically reads the circular buffer of trace records. The trace information is compressed, in some instances interpreted to generate additional information, and a summary is written to a file. In addition, the sampling program can also record information from other kernel data structures, such as those interpreted by the *vmstat* program. Data written out to a file is further buffered to minimize I/O load.

Once a trace log has been created, programs that compress and interpret the data may be run to generate graphs showing the data and relationships between traced events and system load.

The trace package was used mainly to investigate the operation of the file system buffer cache. The sampling program maintained a history of read-ahead blocks and used the trace information to calculate, for example, percentage of read-ahead blocks used.

2.4. Benchmark programs

Benchmark programs were used in two ways. First, a suite of programs was constructed to calculate the cost of certain basic system operations. Operations such as system call overhead and context switching time are critically important in evaluating the overall performance of a system. Because of the drastic changes in the system between 4.1BSD and 4.2BSD, it was important to verify the overhead of these low level operations had not changed appreciably.

The second use of benchmarks was in exercising suspected bottlenecks. When we suspected a specific problem with the system, a small benchmark program was written to repeatedly use the facility. While these benchmarks are not useful as a general tool they can give quick feedback on whether a hypothesized improvement is really having an effect. It is important to realize that the only real assurance that a change has a beneficial effect is through long term measurements of general timesharing. We have numerous examples where a benchmark program suggests vast improvements while the change in the long term system performance is negligible, and conversely examples in which the benchmark program run more slowly, but the long term system performance improves significantly.

3. Results of our observations

When 4.2BSD was first installed on several large timesharing systems the degradation in performance was significant. Informal measurements showed 4.2BSD providing 80% of the throughput of 4.1BSD (based on load averages observed under a normal timesharing load). Many of the initial problems found were because of programs that were not part of 4.1BSD. Using the techniques described in the previous section and standard process profiling several problems were identified. Later work concentrated on the operation of the kernel itself. In this section we discuss the problems uncovered; in the next section we describe the changes made to the system.

3.1. User programs

3.1.1. Mail system

The mail system was the first culprit identified as a major contributor to the degradation in system performance. At Lucasfilm the mail system is heavily used on one machine, a VAX-11/780 with eight megabytes of memory. Message traffic is usually between users on the same machine and ranges from person-to-person telephone messages to per-organization distribution lists. After conversion to 4.2BSD, it was immediately noticed that mail to distribution lists of 20 or more people caused the system load to jump

---

The standard trace facilities distributed with 4.2 differ slightly from those described here. The time stamp in the distributed system is calculated from the kernel's time of day variable instead of the VAX hardware register, and the buffer cache trace points do not record the transfer size.

During part of these observations the machine had only four megabytes of memory.

by anywhere from 3 to 6 points. The number of processes spawned by the *sendmail* program and the messages sent from *sendmail* to the system logging process, *syslog*, generated significant load both from their execution and their interference with basic system operation. The number of context switches and disk transfers often doubled while *sendmail* operated; the system call rate jumped dramatically. System accounting information consistently showed *sendmail* as the top cpu user on the system.

3.1.2. Network servers

The network services provided in 4.2BSD add new capabilities to the system, but are not without cost. The system uses one daemon process to accept requests for each network service provided. The presence of many such daemons increases the numbers of active processes and files, and requires a larger configuration to support the same number of users. The overhead of the routing and status updates can consume several percent of the cpu. Remote logins and shells incur more overhead than their local equivalents. For example, a remote login uses three processes and a pseudo-terminal handler in addition to the local hardware terminal handler. When using a screen editor, sending and echoing a single character involves four processes on two machines. The additional processes, context switching, network traffic, and terminal handler overhead can roughly triple the load presented by one local terminal user.

3.2. System overhead

To measure the costs of various functions in the kernel, a profiling system was run for a 17 hour period on one of our general timesharing machines. While this is not as reproducible as a synthetic workload, it certainly represents a realistic test. This test was run on several occasions over a three month period. Despite the long period of time that elapsed between the test runs the shape of the profiles, as measured by the number of times each system call entry point was called, were remarkably similar.

These profiles turned up several bottlenecks that are discussed in the next section. Several of these were new to 4.2BSD, but most were caused by overloading of mechanisms which worked acceptably well in previous BSD systems. The general conclusion from our measurements was that the ratio of user to system time had increased from 45% system / 55% user in 4.1BSD to 57% system / 43% user in 4.2BSD.

3.2.1. Micro-operation benchmarks

To compare certain basic system operations between 4.1BSD and 4.2BSD a suite of benchmark programs was constructed and run on a VAX-11/750 with 4.5 megabytes of physical memory and two disks on a MASSBUS controller. Tests were run with the machine operating in single user mode under both 4.1BSD and 4.2BSD. Paging was localized to the drive where the root file system was located.

The benchmark programs were modeled after the Kashtan benchmarks, [Kashtan80], with identical sources compiled under each system. The programs and their intended purpose are described briefly before the presentation of the results. The benchmark scripts were run twice with the results shown as the average of the two runs. The source code for each program and the shell scripts used during the benchmarks are included in the Appendix.

The set of tests shown in Table 1 was concerned with system operations other than paging. The intent of most benchmarks is clear. The result of running *signocsw* is deducted from the *csw* benchmark to calculate the context switch overhead. The *exec* tests use two different jobs to gauge the cost of overlaying a larger program with a smaller one and vice versa. The ‘‘null job’’ and ‘‘big job’’ differ solely in the size of their data segments, 1 kilobyte versus 256 kilobytes. In both cases the text segment of the parent is larger than that of the child. All programs were compiled into the default load format that causes the text segment to be demand paged out of the file system and shared between processes.

The results of these tests are shown in Table 2. If the 4.1BSD results are scaled to reflect their being run on a VAX-11/750, they correspond closely to those found in [Joy80].

In studying the measurements we found that the basic system call and context switch overhead did

---

These tests should also have measured the cost of expanding the text segment; unfortunately time did not permit running additional tests.

We assume that a VAX-11/750 runs at 60% of the speed of a VAX-11/780 (not considering floating point operations).

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

Table 1. Kernel Benchmark programs.

not change significantly between 4.1BSD and 4.2BSD. The *signocsw* results were caused by the changes to the *signal* interface, resulting in an additional subroutine invocation for each call, not to mention additional complexity in the system’s implementation.

The times for the use of pipes are significantly higher under 4.2BSD because of their implementation on top of the interprocess communication facilities. Under 4.1BSD pipes were implemented without the complexity of the socket data structures and with simpler code. Further, while not obviously a factor here, 4.2BSD pipes have less system buffer space provided them than 4.1BSD pipes.

The *exec* tests shown in Table 2 were performed with 34 bytes of environment information under 4.1BSD and 40 bytes under 4.2BSD. To figure the cost of passing data through the environment, the execs0null and execs1knull tests were rerun with 1065 additional bytes of data. The results are show in Table 3.

Test	Real		User		System	
	4.1	4.2	4.1	4.2	4.1	4.2
execs0null	197.0	229.0	4.1	2.6	167.8	212.3
execs1knull	199.0	230.0	4.2	2.6	170.4	214.9

Table 3. Benchmark results with ‘‘large’’ environment (all times in seconds).

These results show that passing argument data is significantly slower than under 4.1BSD: 121 ms/byte versus 93 ms/byte. Even using this factor to adjust the basic overhead of an *exec* system call, this facility is

Berkeley Software Distribution UNIX Systems									
Test	Elapsed Time			User Time			System Time		
	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. Kernel Benchmark results (all times in seconds).

more costly under 4.2BSD than under 4.1BSD.

3.2.2. Path name translation

The single most expensive function performed by the kernel is path name translation. This has been true in almost every UNIX kernel [Mosher80]; we find that our general time sharing systems do about 500,000 name translations per day.

Name translations became more expensive in 4.2BSD for several reasons. The single most expensive addition was the symbolic link. Symbolic links have the effect of increasing the average number of components in path names to be translated. As an insidious example, consider the system manager that decides to change /tmp to be a symbolic link to /usr/tmp. A name such as /tmp/tmp1234 that previously required two component translations, now requires four component translations plus the cost of reading the contents of the symbolic link.

The new directory format also changes the characteristics of name translation. The more complex format requires more computation to determine where to place new entries in a directory. Conversely the additional information allows the system to only look at active entries when searching, hence searches of directories that had once grown large but currently have few active entries are checked quickly. The new format also stores the length of each name so that costly string comparisons are only done on names that are the same length as the name being sought.

The net effect of the changes is that the average time to translate a path name in 4.2BSD is 24.2 milliseconds, representing 40% of the time processing system calls, that is 19% of the total cycles in the kernel, or 11% of all cycles executed on the machine. The times are shown in Table 4. We have no comparable times for *namei* under 4.1 though they are certain to be significantly less.

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

Table 4. Call times for *namei* in 4.2BSD.

3.2.3. Clock processing

Nearly 25% of the time spent in the kernel is spent in the clock processing routines. (This is a clear indication that to avoid sampling bias when profiling the kernel with our tools we need to drive them from an independent clock.) These routines are responsible for implementing timeouts, scheduling the processor, maintaining kernel statistics, and tending various hardware operations such as draining the terminal input silos. Only minimal work is done in the hardware clock interrupt routine (at high priority), the rest is performed (at a lower priority) in a software interrupt handler scheduled by the hardware interrupt handler. In the worst case, with a clock rate of 100 Hz and with every hardware interrupt scheduling a software interrupt, the processor must field 200 interrupts per second. The overhead of simply trapping and returning is 3% of the machine cycles, figuring out that there is nothing to do requires an additional 2%.

3.2.4. Terminal multiplexors

The terminal multiplexors supported by 4.2BSD have programmable receiver silos that may be used in two ways. With the silo disabled, each character received causes an interrupt to the processor. Enabling the receiver silo allows the silo to fill before generating an interrupt, allowing multiple characters to be read for each interrupt. At low rates of input, received characters will not be processed for some time unless the silo is emptied periodically. The 4.2BSD kernel uses the input silos of each terminal multiplexor, and empties each silo on each clock interrupt. This allows high input rates without the cost of per-character interrupts while assuring low latency. However, as character input rates on most machines are usually low (about 25 characters per second), this can result in excessive overhead. At the current clock rate of 100 Hz, a machine with 5 terminal multiplexors configured makes 500 calls to the receiver interrupt routines per second. In addition, to achieve acceptable input latency for flow control, each clock interrupt must schedule a software interrupt to run the silo draining routines.
This implies that the worst case estimate for clock processing is the basic overhead for clock processing.

3.2.5. Process table management

In 4.2BSD there are numerous places in the kernel where a linear search of the process table is performed:

- in *exit* to locate and wakeup a process's parent;
- in *wait* when searching for **ZOMBIE** and **STOPPED** processes;
- in *fork* when allocating a new process table slot and counting the number of processes already created by a user;
- in *newproc*, to verify that a process id assigned to a new process is not currently in use;
- in *kill* and *gsignal* to locate all processes to which a signal should be delivered;
- in *schedcpu* when adjusting the process priorities every second; and

---

It is not possible to check the input silos at the time of the actual clock interrupt without modifying the terminal line disciplines, as the input queues may not be in a consistent state .

- in *sched* when locating a process to swap out and/or swap in.

These linear searches can incur significant overhead. The rule for calculating the size of the process table is:

$$\text{nproc} = 20 + 8 * \text{maxusers}$$

that means a 48 user system will have a 404 slot process table. With the addition of network services in 4.2BSD, as many as a dozen server processes may be maintained simply to await incoming requests. These servers are normally created at boot time which causes them to be allocated slots near the beginning of the process table. This means that process table searches under 4.2BSD are likely to take significantly longer than under 4.1BSD. System profiling shows that as much as 20% of the time spent in the kernel on a loaded system (a VAX-11/780) can be spent in *schedcpu* and, on average, 5-10% of the kernel time is spent in *schedcpu*. The other searches of the proc table are similarly affected. This shows the system can no longer tolerate using linear searches of the process table.

3.2.6. File system buffer cache

The trace facilities described in section 2.3 were used to gather statistics on the performance of the buffer cache. We were interested in measuring the effectiveness of the cache and the read-ahead policies. With the file system block size in 4.2BSD four to eight times that of a 4.1BSD file system, we were concerned that large amounts of read-ahead might be performed without being used. Also, we were interested in seeing if the rules used to size the buffer cache at boot time were severely affecting the overall cache operation.

The tracing package was run over a three hour period during a peak mid-afternoon period on a VAX 11/780 with four megabytes of physical memory. This resulted in a buffer cache containing 400 kilobytes of memory spread among 50 to 200 buffers (the actual number of buffers depends on the size mix of disk blocks being read at any given time). The pertinent configuration information is shown in Table 5.

Controller	Drive	Device	File System
DEC MASSBUS	DEC RP06	hp0d	/usr
		hp0b	swap
Emulex SC780	Fujitsu Eagle	hp1a	/usr/spool/news
		hp1b	swap
		hp1e	/usr/src
		hp1d	/u0 (users)
	Fujitsu Eagle	hp2a	/tmp
		hp2b	swap
		hp2d	/u1 (users)
	Fujitsu Eagle	hp3a	/

Table 5. Active file systems during buffer cache tests.

During the test period the load average ranged from 2 to 13 with an average of 5. The system had no idle time, 43% user time, and 57% system time. The system averaged 90 interrupts per second (excluding the system clock interrupts), 220 system calls per second, and 50 context switches per second (40 voluntary, 10 involuntary).

The active virtual memory (the sum of the address space sizes of all jobs that have run in the previous twenty seconds) over the period ranged from 2 to 6 megabytes with an average of 3.5 megabytes. There was no swapping, though the page daemon was inspecting about 25 pages per second.

On average 250 requests to read disk blocks were initiated per second. These include read requests for file blocks made by user programs as well as requests initiated by the system. System reads include requests for indexing information to determine where a file's next data block resides, file system layout maps to allocate new data blocks, and requests for directory contents needed to do path name translations.

On average, an 85% cache hit rate was observed for read requests. Thus only 37 disk reads were initiated per second. In addition, 5 read-ahead requests were made each second filling about 20% of the buffer pool. Despite the policies to rapidly reuse read-ahead buffers that remain unclaimed, more than 90% of the read-ahead buffers were used.

These measurements showed that the buffer cache was working effectively. Independent tests have also showed that the size of the buffer cache may be reduced significantly on memory-poor system without severe effects; we have not yet tested this hypothesis [Shannon83].

3.2.7. Network subsystem

The overhead associated with the network facilities found in 4.2BSD is often difficult to gauge without profiling the system. This is because most input processing is performed in modules scheduled with software interrupts. As a result, the system time spent performing protocol processing is rarely attributed to the processes that really receive the data. Since the protocols supported by 4.2BSD can involve significant overhead this was a serious concern. Results from a profiled kernel show an average of 5% of the system time is spent performing network input and timer processing in our environment (a 3Mb/s Ethernet with most traffic using TCP). This figure can vary significantly depending on the network hardware used, the average message size, and whether packet reassembly is required at the network layer. On one machine we profiled over a 17 hour period (our gateway to the ARPANET) 206,000 input messages accounted for 2.4% of the system time, while another 0.6% of the system time was spent performing protocol timer processing. This machine was configured with an ACC LH/DH IMP interface and a DMA 3Mb/s Ethernet controller.

The performance of TCP over slower long-haul networks was degraded substantially by two problems. The first problem was a bug that prevented round-trip timing measurements from being made, thus increasing retransmissions unnecessarily. The second was a problem with the maximum segment size chosen by TCP, that was well-tuned for Ethernet, but was poorly chosen for the ARPANET, where it causes packet fragmentation. (The maximum segment size was actually negotiated upwards to a value that resulted in excessive fragmentation.)

When benchmarked in Ethernet environments the main memory buffer management of the network subsystem presented some performance anomalies. The overhead of processing small ‘‘mbufs’’ severely affected throughput for a substantial range of message sizes. In spite of the fact that most system utilities made use of the throughput optimal 1024 byte size, user processes faced large degradations for some arbitrary sizes. This was specially true for TCP/IP transmissions [Cabrera84, Cabrera85].

3.2.8. Virtual memory subsystem

We ran a set of tests intended to exercise the virtual memory system under both 4.1BSD and 4.2BSD. The tests are described in Table 6. The test programs dynamically allocated a 7.3 Megabyte array (using *sbrk* (2)) then referenced pages in the array either: sequentially, in a purely random fashion, or such that the distance between successive pages accessed was randomly selected from a Gaussian distribution. In the last case, successive runs were made with increasing standard deviations.

Test	Description
seqpage	sequentially touch pages, 10 iterations
seqpage-v	as above, but first make *vadvise* (2) call
randpage	touch random page 30,000 times
randpage-v	as above, but first make *vadvise* call
gausspage.1	30,000 Gaussian accesses, standard deviation of 1
gausspage.10	as above, standard deviation of 10
gausspage.30	as above, standard deviation of 30
gausspage.40	as above, standard deviation of 40
gausspage.50	as above, standard deviation of 50
gausspage.60	as above, standard deviation of 60
gausspage.80	as above, standard deviation of 80
gausspage.inf	as above, standard deviation of 10,000

Table 6. Paging benchmark programs.

The results in Table 7 show how the additional memory requirements of 4.2BSD can generate more work for the paging system. Under 4.1BSD, the system used 0.5 of the 4.5 megabytes of physical memory on the test machine; under 4.2BSD it used nearly 1 megabyte of physical memory. This resulted in more page faults and, hence, more system time. To establish a common ground on which to compare the paging routines of each system, we check instead the average page fault service times for those test runs that had a statistically significant number of random page faults. These figures, shown in Table 8, show no significant difference between the two systems in the area of page fault servicing. We currently have no explanation for the results of the sequential paging tests.

Test	Real		User		System		Page Faults	
	4.1	4.2	4.1	4.2	4.1	4.2	4.1	4.2
seqpage	959	1126	16.7	12.8	197.0	213.0	17132	17113
seqpage-v	579	812	3.8	5.3	216.0	237.7	8394	8351
randpage	571	569	6.7	7.6	64.0	77.2	8085	9776
randpage-v	572	562	6.1	7.3	62.2	77.5	8126	9852
gausspage.1	25	24	23.6	23.8	0.8	0.8	8	8
gausspage.10	26	26	22.7	23.0	3.2	3.6	2	2
gausspage.30	34	33	25.0	24.8	8.6	8.9	2	2
gausspage.40	42	81	23.9	25.0	11.5	13.6	3	260
gausspage.50	113	175	24.2	26.2	19.6	26.3	784	1851
gausspage.60	191	234	27.6	26.7	27.4	36.0	2067	3177
gausspage.80	312	329	28.0	27.9	41.5	52.0	3933	5105
gausspage.inf	619	621	82.9	85.6	68.3	81.5	8046	9650

Table 7. Paging benchmark results (all times in seconds).

Test	Page Faults		PFST	
	4.1	4.2	4.1	4.2
randpage	8085	9776	791	789
randpage-v	8126	9852	765	786
gausspage.inf	8046	9650	848	844

Table 8. Page fault service times (all times in microseconds).

4. Performance Improvements

This section outlines the changes made to the system since the 4.2BSD distribution. The changes reported here were made in response to the problems described in Section 3. 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.

4.1. 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.

The 4.1BSD system used for testing was really a 4.1a system configured with networking facilities and code to support remote file access. The 4.2BSD system also included the remote file access code. Since both systems would be larger than similarly configured ‘‘vanilla’’ 4.1BSD or 4.2BSD system, we consider out conclusions to still be valid.

4.1.1. Name Cacheing

Our initial profiling studies showed that more than one quarter of the time in the system was spent in the pathname translation routine, *namei*, translating path names to inodes[1]. An inspection of *namei* shows that it consists of two nested loops. The outer loop is traversed once per pathname component. The inner loop performs a linear search through a directory looking for a particular pathname component.

Our first idea was to reduce the number of iterations around the inner loop of *namei* by observing that many programs step through a directory performing an operation on each entry in turn. To improve performance for processes doing directory scans, the system keeps track of the directory 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 offset that the previous name was found (instead of from the beginning of the directory). Changing directories invalidates the cache, as does modifying the directory. For programs that step sequentially through a directory with N files, search time decreases from $O(N^2)$ to $O(N)$.

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

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

This change prompted our rerunning a profiled system on a machine containing the new *namei*. The results showed that the time in *namei* dropped by only 2.6 ms/call and still accounted for 36% of the system call time, 18% of the kernel, or about 10% of all the machine cycles. This amounted to a drop in system time from 57% to about 55%. The results are shown in Table 9.

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

Table 9. Call times for *namei* with per-process cache.

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

Frequent requests for a small set of names are best handled with a cache of recent name translations. This has the effect of eliminating the inner loop of *namei*. For each path name component, *namei* first looks in its cache of recent translations for the needed name. If it exists, the directory search can be completely eliminated.

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

[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.
The cache is keyed on a name and the inode and device number of the directory that contains it. Associated with each entry is a pointer to the corresponding entry in the inode table.

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 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 cost of the name cache is about 200 lines of code (about 1.2 kilobytes) and 48 bytes per cache entry. Depending on the size of the system, about 200 to 1000 entries will normally be configured, using 10-50 kilobytes of physical memory. The name cache is resident in memory at all times.

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

Another profiled system was created and measurements were collected over a 17 hour period. These measurements showed a 13 ms/call decrease in *namei*, with *namei* accounting for only 26% of the system call time, 13% of the time in the kernel, or about 7% of all the machine cycles. System time dropped from 55% to about 49%. The results are shown in Table 10.

part	time	% of kernel
self	4.2 ms/call	6.2%
child	4.4 ms/call	6.6%
total	8.6 ms/call	12.8%

Table 10. Call times for *namei* with both caches.

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. Statistics on the performance of both caches show that the large performance improvement is caused by the high hit ratio. 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 13%. While the system wide cache reduces both the amount of time in the routines that *namei* calls as well as *namei* itself (since fewer directories need to be accessed or searched), it is interesting to note that the actual percentage of system time spent in *namei* itself increases even though the actual time per call decreases. This is because less total time is being spent in the kernel, hence a smaller absolute time becomes a larger total percentage.

[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.

4.1.2. Intelligent Auto Siloing

Most 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. At this input rate they are most efficiently handled with interrupt per character mode, since this generates fewer interrupts than draining the input silos of the terminal multiplexors at each clock interrupt. When input is being generated by another machine or a malfunctioning terminal connection, however, the input rate is usually more than 50 characters per second. It is more efficient to use a device's silo input mode, since this generates fewer interrupts than handling each character as a separate interrupt. 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.

We therefore changed the terminal multiplexor handlers to dynamically choose between the use of the silo and the use of per-character interrupts. At low input rates the handler processes characters on an interrupt basis, avoiding the overhead of checking each interface on each clock interrupt. During periods of sustained input, the handler enables the silo and starts a timer to drain input. This timer runs less frequently than the clock interrupts, and is used only when there is a substantial amount of input. The transition from using silos to an interrupt per character is damped to minimize the number of transitions with bursty traffic (such as in network communication). Input characters serve to flush the silo, preventing long latency. By switching between these two modes of operation dynamically, the overhead of checking the silos is incurred only when necessary.

In addition to the savings in the terminal handlers, the clock interrupt routine is no longer required to schedule a software interrupt after each hardware interrupt to drain the silos. The software-interrupt level portion of the clock routine is only needed when timers expire or the current user process is collecting an execution profile. Thus, the number of interrupts attributable to clock processing is substantially reduced.

4.1.3. 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.

4.1.4. 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.

4.1.5. 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.

4.1.6. 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.

4.1.7. 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 been 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.

The buffer management strategy implemented by *sosend* has been changed to make better use of the increased size of the socket buffers and a better tuned delayed acknowledgement algorithm. Routing has been modified to include a one element cache of the last route computed. Multiple messages send with the same destination now require less processing. Figures 1 and 2 present typical throughput rates that user processes in 4.3BSD systems may expect when run under light load. In [Cabrera85] we documented the performance degradation due to load in either the sender host, receiver host, or ether. Any CPU contention degrades substantially the throughput achievable by user processes. We have observed empty VAX

11/750s using up to 90% of their cycles transmitting network messages.

Figure 1. (I owe it. lfc)

Figure 2. (I owe it. lfc)

4.1.8. 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%.

4.1.9. 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.

4.1.10. 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%.

4.1.11. 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 Table 11. 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.

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.

Alternative C Language Code Optimizations					
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		

Table 11. Alternative inline code expansions.

4.2. 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 was small. By contrast, many of the libraries and utilities had never been tuned. For example, we 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.

4.2.1. 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.

4.2.2. Buffered I/O

The new filesystem with its larger block sizes allows better performance, but it is possible to degrade system performance by performing numerous small transfers rather than using appropriately-sized buffers. The standard I/O library automatically determines the optimal buffer size for each file. Some C library routines and commonly-used programs use low-level I/O or their own buffering, however. 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.

The standard error output has traditionally been unbuffered to prevent delay in presenting the output to the user, and to prevent it from being lost if buffers are not flushed. The inordinate expense of sending single-byte packets through the network led us to impose a buffering scheme on the standard error stream. Within a single call to *fprintf*, all output is buffered temporarily. Before the call returns, all output is flushed and the stream is again marked unbuffered. As before, the normal block or line buffering mechanisms can be used instead of the default behavior.

It is possible for programs with good intentions to unintentionally defeat the standard I/O library's choice of I/O buffer size by using the *setbuf* call to assign an output buffer. Because of portability requirements, the default buffer size provided by *setbuf* is 1024 bytes; this can lead, once again, to added overhead. One such program with this problem was *cat*; there are undoubtedly other standard system utilities with similar problems as the system has changed much since they were originally written.

4.2.3. Mail System

The problems discussed in section 3.1.1 prompted significant work on the entire mail system. The first problem identified was a bug in the *syslog* program. The mail delivery program, *sendmail* logs all mail transactions through this process with the 4.2BSD interprocess communication facilities. *Syslog* then records the information in a log file. Unfortunately, *syslog* was performing a *sync* operation after each message it received, whether it was logged to a file or not. This wreaked havoc on the effectiveness of the buffer cache and explained, to a large extent, why sending mail to large distribution lists generated such a heavy load on the system (one syslog message was generated for each message recipient causing almost a continuous sequence of sync operations).

The hashed data base files were installed in all mail programs, resulting in a order of magnitude speedup on large distribution lists. The code in */bin/mail* that notifies the *comsat* program when mail has been delivered to a user was changed to cache host table lookups, resulting in a similar speedup on large distribution lists.

Next, the file locking facilities provided in 4.2BSD, *flock* (2), were used in place of the old locking mechanism. 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. This yielded another 10% cut in the basic overhead of delivering mail. Extensive profiling and tuning of *sendmail* and compiling it without debugging code reduced the overhead by another 20%.

4.2.4. 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. Rather than having many servers started at boot time, a single server, *inetd* was substituted. This process reads a simple configuration file that specifies the services the system is willing to support and listens for service requests on each service's Internet port. When a client requests service the appropriate server is created and passed a service connection as its standard input. Servers that require the identity of their client may use the *getpeername* system call; likewise *getsockname* may be used to find out a server's local address without consulting data base files. This scheme is attractive for several reasons:

- it eliminates as many as a dozen processes, easing system overhead and allowing the file and text tables to be made smaller,
- servers need not contain the code required to handle connection queueing, simplifying the programs, and
- installing and replacing servers becomes simpler.

With an increased numbers of networks, both local and external to Berkeley, we found that the overhead of the routing process was becoming inordinately high. Several changes were made in the routing daemon to reduce this load. Routes to external networks are no longer exchanged by routers on the internal machines, only a route to a default gateway. This reduces the amount of network traffic and the time required to process routing messages. In addition, the routing daemon was profiled and functions responsible for large amounts of time were optimized. The major changes were a faster hashing scheme, and inline expansions of the ubiquitous byte-swapping functions.

Under certain circumstances, when output was blocked, attempts by the remote login process to send output to the user were rejected by the system, although a prior *select* call had indicated that data could be sent. This resulted in continuous attempts to write the data until the remote user restarted output. This problem was initially avoided in the remote login handler, and the original problem in the kernel has since been corrected.

4.2.5. 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.

4.2.6. 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.

5. 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.

5.1. 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.

5.1.1. 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.

5.1.2. 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.

5.1.3. 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*

† Ethernet is a trademark of Xerox.

and *setsockopt*, are now interfaced to the protocol level allowing protocol-specific options to be manipulated by the user.

5.1.4. 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.

5.1.5. 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 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.

5.1.6. 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 do such simple tasks 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. Standard configurations support data and stack segments of 17, 33 and 64 megabytes.

5.1.7. 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.

5.1.8. 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.

5.1.9. 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, 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.

5.1.10. 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.

5.1.11. 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.

5.2. 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.

5.2.1. 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 name server operates in accordance with Internet standards for service on the ARPANET [Mockapetris83].

5.2.2. 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.

The new utility *timed* keeps the time on a group of cooperating machines (within a single LAN) 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.

5.2.3. 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.

5.2.4. 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.

6. Security Tightening

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

6.1. 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.

6.2. 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.

7. Conclusions

4.2BSD, while functionally superior to 4.1BSD, lacked much of the performance tuning required of a good system. We found that the distributed system spent 10-20% more time in the kernel than 4.1BSD. This added overhead combined with problems with several user programs severely limited the overall performance of the system in a general timesharing environment.

Changes made to the system since the 4.2BSD distribution have eliminated most of the added system overhead by replacing old algorithms or introducing additional cacheing schemes. The combined caches added to the name translation process reduce the average cost of translating a pathname to an inode by more than 50%. These changes reduce the percentage of time spent running in the system by nearly 9%.

The use of silo input on terminal ports only when necessary has allowed the system to avoid a large amount of software interrupt processing. Observations show that the system is forced to field about 25% fewer interrupts than before.

The kernel changes, combined with many bug fixes, make the system much more responsive in a general timesharing environment. The 4.3BSD Berkeley UNIX system now appears capable of supporting loads at least as large as those supported under 4.1BSD while providing all the new interprocess communication, networking, and file system facilities.

Acknowledgements

We would like to thank Robert Elz for sharing his ideas and his code for cacheing system wide names and searching the process table. We thank Alan Smith for initially suggesting the use of a capability based cache. We also acknowledge George Goble who dropped many of our changes into his production system and reported back fixes to the disasters that they caused. The buffer cache read-ahead trace package was based on a program written by Jim Lawson. Ralph Campbell implemented several of the C library changes. The original version of the Internet daemon was written by Bill Joy. In addition, we would like to thank the many other people that contributed ideas, information, and work while the system was undergoing change.

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.

[Joy80] Joy, William, ‘‘Comments on the performance of UNIX on the VAX’’, Computer System Research Group, U.C. Berkeley. April 1980.

[Kashtan80] Kashtan, David L., ‘‘UNIX and VMS, Some Performance Comparisons’’, SRI International. February 1980.

[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.

[McKusick85] M. Kirk McKusick, Mike Karels, and Samual Leffler, ‘‘Performance Improvements and Functional Enhancements in 4.3BSD’’ *Proceedings of the Portland Usenix Conference*, pp 519-531, June 1985.

[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.

[Mosher80] Mosher, David, ‘‘UNIX Performance, an Introspection’’, Presented at the Boulder, Colorado Usenix Conference, January 1980. Copies of the paper are available from Computer System Research Group, U.C. Berkeley.

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

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

[Shannon83] Shannon, W., private communication, July 1983

[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.

[Cabrera84] Luis Felipe Cabrera, Eduard Hunter, Michael J. Karels, and David Mosher, ‘‘A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX 4.2BSD,’’ Research Report No. UCB/CSD 84/217, University of California, Berkeley, December 1984.

[Cabrera85] Luis Felipe Cabrera, Michael J. Karels, and David Mosher, ‘‘The Impact of Buffer Management on Networking Software Performance in Berkeley UNIX 4.2BSD: A Case Study,’’ Proceedings of the Summer Usenix Conference, Portland, Oregon, June 1985, pp. 507-517.

Appendix A – Benchmark sources

The programs shown here run under 4.2 with only routines from the standard libraries. When run under 4.1 they were augmented with a *getpagesize* routine and a copy of the *random* function from the C library. The *vforks* and *vexecs* programs are constructed from the *forks* and *execs* programs, respectively, by substituting calls to *fork* with calls to *vfork*.

syscall

```
/*
 * System call overhead benchmark.
 */
main(argc, argv)                                                        main
        char *argv[];
{
        register int ncalls;

        if (argc < 2) {
                printf("usage: %s #syscalls\n", argv[0]);
                exit(1);
        }
        ncalls = atoi(argv[1]);
        while (ncalls-- > 0)
                (void) getpid();
}
```

csw

```
/*
 * Context switching benchmark.
 *
 * Force system to context switch 2*nsigs
 * times by forking and exchanging signals.
 * To calculate system overhead for a context
 * switch, the signocsw program must be run
 * with nsigs.  Overhead is then estimated by
 *         t1 = time csw <n>
 *         t2 = time signocsw <n>
 *         overhead = t1 - 2 * t2;
 */
#include <signal.h>

int     sigsub();
int     otherpid;
int     nsigs;

main(argc, argv)                                                        main
        char *argv[];
{
        int pid;

        if (argc < 2) {
                printf("usage: %s nsignals\n", argv[0]);
                exit(1);
        }
        nsigs = atoi(argv[1]);
        signal(SIGALRM, sigsub);
```

```
	otherpid = getpid();
	pid = fork();
	if (pid != 0) {
		otherpid = pid;
		kill(otherpid, SIGALRM);
	}
	for (;;)
		sigpause(0);
}
```

sigsub

```
sigsub()
{

	signal(SIGALRM, sigsub);
	kill(otherpid, SIGALRM);
	if (—nsigs <= 0)
		exit(0);
}
```

signocsw

```
/*
 * Signal without context switch benchmark.
 */
#include <signal.h>

int	pid;
int	nsigs;
int	sigsub();
```

main

```
main(argc, argv)
	char *argv[];
{
	register int i;

	if (argc < 2) {
		printf("usage: %s nsignals\n", argv[0]);
		exit(1);
	}
	nsigs = atoi(argv[1]);
	signal(SIGALRM, sigsub);
	pid = getpid();
	for (i = 0; i < nsigs; i++)
		kill(pid, SIGALRM);
}
```

sigsub

```
sigsub()
{

	signal(SIGALRM, sigsub);
}
```

pipeself

```
/*
 * IPC benchmark,
 * write to self using pipes.
 */

main(argc, argv)                                              main
        char *argv[];
{
        char buf[512];
        int fd[2], msgsize;
        register int i, iter;

        if (argc < 3) {
                printf("usage: %s iterations message–size\n", argv[0]);
                exit(1);
        }
        argc––, argv++;
        iter = atoi(*argv);
        argc––, argv++;
        msgsize = atoi(*argv);
        if (msgsize > sizeof (buf) || msgsize <= 0) {
                printf("%s: Bad message size.\n", *argv);
                exit(2);
        }
        if (pipe(fd) < 0) {
                perror("pipe");
                exit(3);
        }
        for (i = 0; i < iter; i++) {
                write(fd[1], buf, msgsize);
                read(fd[0], buf, msgsize);
        }
}
```

pipediscard

```
/*
 * IPC benchmarkl,
 * write and discard using pipes.
 */

main(argc, argv)                                              main
        char *argv[];
{
        char buf[512];
        int fd[2], msgsize;
        register int i, iter;

        if (argc < 3) {
                printf("usage: %s iterations message–size\n", argv[0]);
                exit(1);
        }
        argc––, argv++;
        iter = atoi(*argv);
```

```
        argc--, argv++;
        msgsize = atoi(*argv);
        if (msgsize > sizeof (buf) || msgsize <= 0) {
                printf("%s: Bad message size.\n", *argv);
                exit(2);
        }
        if (pipe(fd) < 0) {
                perror("pipe");
                exit(3);
        }
        if (fork() == 0)
                for (i = 0; i < iter; i++)
                        read(fd[0], buf, msgsize);
        else
                for (i = 0; i < iter; i++)
                        write(fd[1], buf, msgsize);
}
```

pipeback

```
/*
 * IPC benchmark,
 * read and reply using pipes.
 *
 * Process forks and exchanges messages
 * over a pipe in a request–response fashion.
 */

main(argc, argv)                                                    main
        char *argv[];
{
        char buf[512];
        int fd[2], fd2[2], msgsize;
        register int i, iter;

        if (argc < 3) {
                printf("usage: %s iterations message–size\n", argv[0]);
                exit(1);
        }
        argc--, argv++;
        iter = atoi(*argv);
        argc--, argv++;
        msgsize = atoi(*argv);
        if (msgsize > sizeof (buf) || msgsize <= 0) {
                printf("%s: Bad message size.\n", *argv);
                exit(2);
        }
        if (pipe(fd) < 0) {
                perror("pipe");
                exit(3);
        }
        if (pipe(fd2) < 0) {
                perror("pipe");
                exit(3);
        }
```

```
		if (fork() == 0)
			for (i = 0; i < iter; i++) {
				read(fd[0], buf, msgsize);
				write(fd2[1], buf, msgsize);
			}
		else
			for (i = 0; i < iter; i++) {
				write(fd[1], buf, msgsize);
				read(fd2[0], buf, msgsize);
			}
}
```

forks

```
/*
 * Benchmark program to calculate fork+wait
 * overhead (appr
```