Measuring and Improving the Performance of 4.2BSD

Sam Leffler

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

Mike Karels
M. Kirk McKusick

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

ABSTRACT

The 4.2 Berkeley Software Distribution of UNIX† for the VAX‡ has several problems that can 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. Suggestions for further performance improvements are given.

† UNIX is a trademark of Bell Laboratories.
‡ VAX, MASSBUS, UNIBUS, and DEC are trademarks of Digital Equipment Corporation.
This work was sponsored in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

TABLE OF CONTENTS

1. Introduction

The 4.2 Berkeley Software Distribution of UNIX for the VAX has added many new capabilities that were previously unavailable under UNIX. Many new data structures have been added to the system to support these new capabilities. In addition, many of the existing data structures and algorithms have been put to new uses or their old functions placed under increased demand. The effect of these changes is that mechanisms that were well tuned under 4.1BSD no longer provide adequate performance for 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. Thus rather than using simulated workloads, we have sought to analyse our tuning efforts under realistic conditions.

2. Observation techniques

There are many tools available for observing 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, 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 30-35 tps in practice), and the user cpu utilization (us) should be high (above 60%).

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 1" 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 nonbuffered input/output, then the system time may go high (60-70% 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 indicate we average about 60 context switches and interrupts per second and about 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 system performance and pregnant 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 indicate 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 very 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 which 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 which 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.[1]

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 which compress and interpret the data may be run to generate graphs showing the data and/or 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. Due to 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 4.2 was first installed on several large timesharing systems the degradation in performance was significant. Informal measurements indicated 4.2 performed at about 80% of that of 4.1 (based on load averages observed under a normal timesharing load). Many of the initial problems found were due to programs which were not part of 4.1. 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 one of the first culprits identified as a major contributor to the degradation in system performance. At Lucasfilm the mail system is very heavily used on one machine, a VAX-11/780 with eight megabytes of memory.[2] Message traffic is usually between users on the same machine and ranges from person-to-person telephone messages to per-organization

[1] The standard trace facilities distributed with 4.2 actually 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.

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

distribution lists. After conversion to 4.2, it was immediately noticed that mail to distribution lists of 20 or more people caused the system load to jump 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 4.2 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 a number of 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 be appreciable, on the order of several percent of the cpu. Remote logins and shells incur more overhead than their local equivalents. For example, a remote login utilizes 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 an overloading of a mechanism that 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 4.1 and 4.2. 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 prior to 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 Appendix A.

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.[3] All programs were

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

compiled into the default load format which causes the text segment to be demand paged out of the file system and shared between processes.

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
execs1knull	sbrk(1024), fault page, fork-exec "null job"-exit-wait 1,000 times
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. Benchmark programs.

The results of these tests are shown in Table 2. If the 4.1 results are scaled to reflect their being run on a VAX-11/750, they correspond closely to those indicated in [Joy80].[4]

In studying the times we find the basic system call and context switching overhead have not changed significantly between 4.1 and 4.2. The *signocsw* results reflect the fact that the *signal* interface has changed, 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 4.2 due to their implementation on top of the interprocess communication facilities. Under 4.1 pipes were implemented without the complexity of the socket data structures and with simpler code. Further, while not obviously a factor here, 4.2 pipes have less system buffer space provided them than 4.1 pipes.

The exec tests shown in Table 2 were performed with 34 bytes of environment information under 4.1 and 40 bytes under 4.2. 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. These results indicate passing argument data is significantly higher than under 4.1: 121 ms/byte versus 93 ms/byte. Even using this factor to adjust the basic overhead of an *exec* system call, this facility is more costly under 4.2 than under 4.1.

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

Test	Real		User		System	
	4.1	4.2	4.1	4.2	4.1	4.2
syscall	28.0	29.0	4.5	5.3	23.9	23.7
csw	45.0	44.0	3.5	3.7	19.5	18.7
signocsw	16.5	22.0	1.9	2.3	14.6	20.3
pipeself4	21.5	29.0	1.1	1.1	20.1	28.0
pipeself512	47.5	59.0	1.2	1.2	46.1	58.3
pipediscard4	32.0	42.0	3.2	3.7	15.5	18.8
pipediscard512	61.0	76.0	3.1	2.1	29.7	36.4
pipeback4	57.0	75.0	2.9	3.2	25.1	34.2
pipeback512	110.0	138.0	3.1	3.4	52.2	65.7
forks0	37.5	41.0	0.5	0.3	34.5	37.6
forks1k	40.0	43.0	0.4	0.3	36.0	38.8
forks100k	217.5	223.0	0.7	0.6	214.3	218.4
vforks0	34.5	37.0	0.5	0.6	27.3	28.5
vforks1k	35.0	37.0	0.6	0.8	27.2	28.6
vforks100k	35.0	37.0	0.6	0.8	27.6	28.9
execs0null	97.5	92.0	3.8	2.4	68.7	82.5
execs1knull	99.0	100.0	4.1	1.9	70.5	86.8
execs100knull	283.5	278.0	4.8	2.8	251.9	269.3
vexecs0null	100.0	92.0	5.1	2.7	63.7	76.8
vexecs1knull	100.0	91.0	5.2	2.8	63.2	77.1
vexecs100knull	100.0	92.0	5.1	3.0	64.0	77.7
execs0big	129.0	201.0	4.0	3.0	102.6	153.5
execs1kbig	130.0	202.0	3.7	3.0	104.7	155.5
execs100kbig	318.0	385.0	4.8	3.1	286.6	339.1
vexecs0big	128.0	200.0	4.6	3.5	98.5	149.6
vexecs1kbig	125.0	200.0	4.7	3.5	98.9	149.3
vexecs100kbig	126.0	200.0	4.2	3.4	99.5	151.0

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

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).

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, which 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*.

3.2.3. Clock processing

Nearly 25% of the time spent in the kernel is spent in the clock processing routines. 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 quite 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.[5] This implies that the worst case estimate for clock processing is, in fact, the basic overhead for clock processing.

3.2.5. Process table management

In 4.2 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;

[5] 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 *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
- 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:

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

which means a 48 user system will have a 404 slot process table. With the addition of network services in 4.2, 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 4.2 are likely to take significantly longer than under 4.1. System profiling indicates 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 indicates 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 4.2 four to eight times that of a 4.1 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 indicated that the buffer cache was working effectively. Independent tests have also indicated that the size of the buffer cache may be reduced significantly on memory-poor system without severe effects; we have not, as yet, tested this conjecture [Shannon83].

3.2.7. Network subsystem

The overhead associated with the network facilities found in 4.2 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 which actually receive the data. Since the protocols supported by 4.2 can involve significant overhead this was a serious concern. Results from a profiled kernel indicate 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, which was well-tuned for Ethernet, but which was poorly chosen for the ARPANET, where it causes packet fragmentation. (The maximum segment size was actually negotiated upwards to a value which resulted in excessive fragmentation.)

3.2.8. Virtual memory subsystem

We ran a set of tests intended to exercise the virtual memory system under both 4.1 and 4.2. 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.

The results in Table 7 show how the additional memory requirements of 4.2 can generate more work for the paging system. Under 4.1, the system used 0.5 of the 4.5 megabytes of physical memory on the test machine; under 4.2 it used nearly 1 megabyte of physical memory.[6] This

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

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.

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 which had a statistically significant number of random page faults. These figures, shown in Table 8, indicate 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).

valid.

4. System Changes

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

4.1. User programs

Several changes were made in the C library which affected many user programs.

4.1.1. Hashed data bases

A new version of the *dbm* (3X) library was created which supported multiple open data base files per process. This was then used to rewrite the access routines for the password and host data bases. These changes had most significant impact on the performance of the mail system, particularly in a large user and/or host environment (e.g. the ARPANET).

4.1.2. Buffering 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. One such problem was found in the *ttyslot* library function, which read from the *ttys* file one character at a time. This was changed so that reads were buffered. Two other problems were found in the loader and the assembler, both of which used their own buffering schemes. One kilobyte buffers, as opposed to buffers equal in size to the the filesystem block size were discovered. Both have been changed to choose their buffer sizes appropriately for the underlying filesystem.

The standard error output has traditionally been unbuffered in order 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. Due to 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.

4.1.3. Mail system

The problems indicated 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 4.2 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* which 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 4.2,

flock (2), were used in place of the old locking mechanism. This yielded another 10% cut in the basic overhead of delivering mail. Finally *sendmail* was compiled without debugging code, reducing the overhead by another 5%.

The resultant system, while much faster than that originally distributed with 4.2, was still too costly to run at Lucasfilm. This forced the *sendmail* program to be replaced with a simpler delivery system, *sm*, which is 2-3 times faster than the revamped *sendmail* [Ostby84]. The speed is gained through:

- smaller code (the work performed by *sendmail* is distributed among many programs),
- no configuration file (everything is compiled in),
- simpler address parsing,
- buffering small mail messages in memory rather than rereading a temporary file[7],
- performing local mail delivery directly, and
- performing fewer forks.

In addition *sm* logs only critical errors.

4.1.4. Network servers

The overhead generated by having one server process always present listening for each service caused a redesign of the basic mechanism by which a server program is created. Rather than having many servers started at boot time, a single server, *inetd* was substituted. This process reads a simple configuration file which 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 which 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 very 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/or 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.

[7] This can result in messages being lost if the system crashes while the message is buffered in memory. Insuring this does not happen is possible with the proper *sendmail* configuration, but is costly since it requires several disk writes per message.

4.2. Kernel changes

Several changes were made to speed up the bottlenecks discovered in the kernel.

4.2.1. Name cacheing

The system measurements collected indicated the pathname translation routine, *namei*, was clearly worth optimizing. 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 use the fact that many programs step through a directory performing an operation on each entry in turn. This caused us to modify *namei* to cache the directory offset of the last pathname component looked up by a process. The cached offset is then used as the point at which a search in the same directory begins. Changing directories invalidates the cache, as does modifying the directory. For programs which 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 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 indicated 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 relatively 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.

Most missed names were caused by path name components other than the last. Thus Robert Elz introduced a cache of most recent name translations[8]. This had the effect of short circuiting the outer 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. If the name is not recognized, then the per-process cache may still be useful in reducing the directory search time. The two cacheing schemes complement each other well.

The cost of the name cache is about 200 lines of code (about 1.2 kilobytes) and 44 bytes per cache entry. Depending on the size of the system, about 200 to 1000 entries will normally be configured, using 10-44 kilobytes of physical memory. The name cache is resident in memory at all times.

[8] 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.

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 indicated a 6 ms/call decrease in *namei*, with *namei* accounting for only 31% of the system call time, 16% 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	9.5 ms/call	9.6%
child	6.1 ms/call	6.1%
total	15.6 ms/call	15.7%

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

Statistics on the performance of both caches indicate the large performance improvement is caused by the high hit ratio. On the profiled system a 60% hit rate was observed in the system wide cache. This, coupled with the 25% hit rate in the per-process offset cache yielded an effective cache hit rate of 85%. 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.

4.2.2. Auto-siloing terminal input

We observed a low rate of terminal input on most of our systems, which motivated us to re-enable interrupts on a per-character basis. This would allow us to save the high overhead incurred by the system in draining the input silos of the terminal multiplexors at each clock interrupt. Unfortunately, this change would result in huge interrupt loads during periods of heavy input from networks, high-speed devices or malfunctioning terminal connections. 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 for the purpose of draining 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. Accordingly, the number of interrupts attributable to clock processing is substantially reduced.

4.2.3. Process table management

As noted in section 3.2.5, the linear searches of the process table can result in significant overhead. Consequently, we incorporated changes made by Robert Elz to eliminate all linear searches of the process table. Three separate linked lists are maintained for all: active (allocated) process table entries, inactive (unallocated) process table entries, and for processes in zombie state. These lists eliminate the most expensive process table searches performed by the system. In addition, pointers in the process structure which maintain related processes in a tree structure and which previously had been maintained but not used, were finally used to eliminate the linear

searches for parent and sibling processes. These changes were incorporated too late for us to accurately measure the reduction in system overhead.

5. Future work

Many areas for further work still exist. There is still a need to reduce the overhead introduced by the revised system call interfaces for pipes and signals, and for existing facilities such as exec. The system wide name cache does not currently support the inclusion of "." and ".."; adding this capability may significantly increase the hit rate. The 100 Hz clock rate needs to be more carefully examined. The initial motivation for this change, to increase the precision of all timing facilities, must be weighed against the basic clock processing overhead. Tom Ferrin has experimented with cutting the clock rate in half with some success [Ferrin84]; it is unclear whether this will result in a significant gain given the changes already made to the clock handling code. Finally, several anomalous test results need to be understood and the late changes to the handling of the process table need to be evaluated.

6. 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 4.1. 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 4.2 distribution have eliminated most of the added system overhead by replacing old algorithms and/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 35%. 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 indicate 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 system now appears capable of supporting loads at least as large as supported under 4.1 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 also acknowledge George Goble who dropped many of our changes into his production system and reported back fixes to the disasters that they caused. Eben Ostby did the work on the mail system. 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

[Ferrin84] Ferrin, Tom, private communication, January 1984.

[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.

[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.

[Ostby84] Ostby, Eben, "SM: A Small Mailer", Lucasfilm TM. April 25, 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

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(argc, argv)                                            main
        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 (approximately).  Process
 * forks and exits while parent waits.
 * The time to run this program is used
 * in calculating exec overhead.
 */

main(argc, argv)                                                    main
        char *argv[];
{
        register int nforks, i;
        char *cp;
        int pid, child, status, brksize;

        if (argc < 2) {
                printf("usage: %s number-of-forks sbrk-size\n", argv[0]);
                exit(1);
        }
        nforks = atoi(argv[1]);
        if (nforks < 0) {
                printf("%s: bad number of forks\n", argv[1]);
                exit(2);
        }
        brksize = atoi(argv[2]);
        if (brksize < 0) {
                printf("%s: bad size to sbrk\n", argv[2]);
                exit(3);
        }
        cp = (char *)sbrk(brksize);
        if ((int)cp == -1) {
                perror("sbrk");
                exit(4);
        }
        for (i = 0; i < brksize; i += 1024)
                cp[i] = i;
        while (nforks-- > 0) {
                child = fork();
                if (child == -1) {
                        perror("fork");
                        exit(-1);
```

```
                }
                if (child == 0)
                        _exit(-1);
                while ((pid = wait(&status)) != -1 && pid != child)
                        ;
        }
        exit(0);
}
```

execs

```
/*
 * Benchmark program to calculate exec
 * overhead (approximately).  Process
 * forks and execs "null" test program.
 * The time to run the fork program should
 * then be deducted from this one to
 * estimate the overhead for the exec.
 */
```

main

```
main(argc, argv)
        char *argv[];
{
        register int nexecs, i;
        char *cp, *sbrk();
        int pid, child, status, brksize;

        if (argc < 3) {
                printf("usage: %s number-of-execs sbrk-size job-name\n",
                    argv[0]);
                exit(1);
        }
        nexecs = atoi(argv[1]);
        if (nexecs < 0) {
                printf("%s: bad number of execs\n", argv[1]);
                exit(2);
        }
        brksize = atoi(argv[2]);
        if (brksize < 0) {
                printf("%s: bad size to sbrk\n", argv[2]);
                exit(3);
        }
        cp = sbrk(brksize);
        if ((int)cp == -1) {
                perror("sbrk");
                exit(4);
        }
        for (i = 0; i < brksize; i += 1024)
                cp[i] = i;
        while (nexecs-- > 0) {
                child = fork();
                if (child == -1) {
                        perror("fork");
                        exit(-1);
                }
```

```
                if (child == 0) {
                        execv(argv[3], argv);
                        perror("execv");
                        _exit(-1);
                }
                while ((pid = wait(&status)) != -1 && pid != child)
                        ;
        }
        exit(0);
}
```

nulljob

```
/*
 * Benchmark "null job" program.
 */

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

        exit(0);
}
```

bigjob

```
/*
 * Benchmark "null big job" program.
 */
/* 250 here is intended to approximate vi's text+data size */
char    space[1024 * 250] = "force into data segment";

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

        exit(0);
}
```

seqpage

```
/*
 * Sequential page access benchmark.
 */
#include <sys/vadvise.h>

char        *valloc();

main(argc, argv)                                                    main
        char *argv[];
{
        register i, niter;
        register char *pf, *lastpage;
        int npages = 4096, pagesize, vflag = 0;
        char *pages, *name;

        name = argv[0];
        argc--, argv++;
again:
        if (argc < 1) {
usage:
                printf("usage: %s [ -v ] [ -p #pages ] niter\n", name);
                exit(1);
        }
        if (strcmp(*argv, "-p") == 0) {
                argc--, argv++;
                if (argc < 1)
                        goto usage;
                npages = atoi(*argv);
                if (npages <= 0) {
                        printf("%s: Bad page count.\n", *argv);
                        exit(2);
                }
                argc--, argv++;
                goto again;
        }
        if (strcmp(*argv, "-v") == 0) {
                argc--, argv++;
                vflag++;
                goto again;
        }
        niter = atoi(*argv);
        pagesize = getpagesize();
        pages = valloc(npages * pagesize);
        if (pages == (char *)0) {
                printf("Can't allocate %d pages (%2.1f megabytes).\n",
                    npages, (npages * pagesize) / (1024. * 1024.));
                exit(3);
        }
        lastpage = pages + (npages * pagesize);
        if (vflag)
                vadvise(VA_SEQL);
        for (i = 0; i < niter; i++)
                for (pf = pages; pf < lastpage; pf += pagesize)
```

```
                        *pf = 1;
}

randpage
/*
 * Random page access benchmark.
 */
#include <sys/vadvise.h>

char    *valloc();
int     rand();

main(argc, argv)                                                    main
        char *argv[];
{
        register int npages = 4096, pagesize, pn, i, niter;
        int vflag = 0, debug = 0;
        char *pages, *name;

        name = argv[0];
        argc--, argv++;
again:
        if (argc < 1) {
usage:
                printf("usage: %s [ -d ] [ -v ] [ -p #pages ] niter\n", name);
                exit(1);
        }
        if (strcmp(*argv, "-p") == 0) {
                argc--, argv++;
                if (argc < 1)
                        goto usage;
                npages = atoi(*argv);
                if (npages <= 0) {
                        printf("%s: Bad page count.\n", *argv);
                        exit(2);
                }
                argc--, argv++;
                goto again;
        }
        if (strcmp(*argv, "-v") == 0) {
                argc--, argv++;
                vflag++;
                goto again;
        }
        if (strcmp(*argv, "-d") == 0) {
                argc--, argv++;
                debug++;
                goto again;
        }
        niter = atoi(*argv);
        pagesize = getpagesize();
        pages = valloc(npages * pagesize);
        if (pages == (char *)0) {
                printf("Can't allocate %d pages (%2.1f megabytes).\n",
```

```
                        npages, (npages * pagesize) / (1024. * 1024.));
                exit(3);
        }
        if (vflag)
                vadvise(VA_ANOM);
        for (i = 0; i < niter; i++) {
                pn = random() % npages;
                if (debug)
                        printf("touch page %d\n", pn);
                pages[pagesize * pn] = 1;
        }
}
```

gausspage

```
/*
 * Random page access with
 * a gaussian distribution.
 *
 * Allocate a large (zero fill on demand) address
 * space and fault the pages in a random gaussian
 * order.
 */

float   sqrt(), log(), rnd(), cos(), gauss();
char    *valloc();
int     rand();

main(argc, argv)                                                    main
        char *argv[];
{
        register int pn, i, niter, delta;
        register char *pages;
        float sd = 10.0;
        int npages = 4096, pagesize, debug = 0;
        char *name;

        name = argv[0];
        argc--, argv++;
again:
        if (argc < 1) {
usage:
                printf(
"usage: %s [ -d ] [ -p #pages ] [ -s standard-deviation ] iterations\n", name);
                exit(1);
        }
        if (strcmp(*argv, "-s") == 0) {
                argc--, argv++;
                if (argc < 1)
                        goto usage;
                sscanf(*argv, "%f", &sd);
                if (sd <= 0) {
                        printf("%s: Bad standard deviation.\n", *argv);
                        exit(2);
                }
```

```
                argc—, argv++;
                goto again;
        }
        if (strcmp(*argv, "-p") == 0) {
                argc—, argv++;
                if (argc < 1)
                        goto usage;
                npages = atoi(*argv);
                if (npages <= 0) {
                        printf("%s: Bad page count.\n", *argv);
                        exit(2);
                }
                argc—, argv++;
                goto again;
        }
        if (strcmp(*argv, "-d") == 0) {
                argc—, argv++;
                debug++;
                goto again;
        }
        niter = atoi(*argv);
        pagesize = getpagesize();
        pages = valloc(npages*pagesize);
        if (pages == (char *)0) {
                printf("Can't allocate %d pages (%2.1f megabytes).\n",
                    npages, (npages*pagesize) / (1024. * 1024.));
                exit(3);
        }
        pn = 0;
        for (i = 0; i < niter; i++) {
                delta = gauss(sd, 0.0);
                while (pn + delta < 0 || pn + delta > npages)
                        delta = gauss(sd, 0.0);
                pn += delta;
                if (debug)
                        printf("touch page %d\n", pn);
                else
                        pages[pn * pagesize] = 1;
        }
}

float
gauss(sd, mean)                                          gauss
        float sd, mean;
{
        register float qa, qb;

        qa = sqrt(log(rnd()) * -2.0);
        qb = 3.14159 * rnd();
        return (qa * cos(qb) * sd + mean);
}

float
rnd()                                                    rnd
```

```
{
        static int seed = 1;
        static int biggest = 0x7fffffff;

        return ((float)rand(seed) / (float)biggest);
}
```

run (shell script)

```
#! /bin/csh -fx
# Script to run benchmark programs.
#
date
make clean; time make
time syscall 100000
time seqpage -p 7500 10
time seqpage -v -p 7500 10
time randpage -p 7500 30000
time randpage -v -p 7500 30000
time gausspage -p 7500 -s 1 30000
time gausspage -p 7500 -s 10 30000
time gausspage -p 7500 -s 30 30000
time gausspage -p 7500 -s 40 30000
time gausspage -p 7500 -s 50 30000
time gausspage -p 7500 -s 60 30000
time gausspage -p 7500 -s 80 30000
time gausspage -p 7500 -s 10000 30000
time csw 10000
time signocsw 10000
time pipeself 10000 512
time pipeself 10000 4
time udgself 10000 512
time udgself 10000 4
time pipediscard 10000 512
time pipediscard 10000 4
time udgdiscard 10000 512
time udgdiscard 10000 4
time pipeback 10000 512
time pipeback 10000 4
time udgback 10000 512
time udgback 10000 4
size forks
time forks 1000 0
time forks 1000 1024
time forks 1000 102400
size vforks
time vforks 1000 0
time vforks 1000 1024
time vforks 1000 102400
countenv
size nulljob
time execs 1000 0 nulljob
time execs 1000 1024 nulljob
time execs 1000 102400 nulljob
time vexecs 1000 0 nulljob
```

```
time vexecs 1000 1024 nulljob
time vexecs 1000 102400 nulljob
size bigjob
time execs 1000 0 bigjob
time execs 1000 1024 bigjob
time execs 1000 102400 bigjob
time vexecs 1000 0 bigjob
time vexecs 1000 1024 bigjob
time vexecs 1000 102400 bigjob
# fill environment with ~ 1024 bytes
setenv a 0123456789012345678901234567890123456789012345678901234567890123456789
setenv b 0123456789012345678901234567890123456789012345678901234567890123456789
setenv c 0123456789012345678901234567890123456789012345678901234567890123456789
setenv d 0123456789012345678901234567890123456789012345678901234567890123456789
setenv e 0123456789012345678901234567890123456789012345678901234567890123456789
setenv f 0123456789012345678901234567890123456789012345678901234567890123456789
setenv g 0123456789012345678901234567890123456789012345678901234567890123456789
setenv h 0123456789012345678901234567890123456789012345678901234567890123456789
setenv i 0123456789012345678901234567890123456789012345678901234567890123456789
setenv j 0123456789012345678901234567890123456789012345678901234567890123456789
setenv k 0123456789012345678901234567890123456789012345678901234567890123456789
setenv l 0123456789012345678901234567890123456789012345678901234567890123456789
setenv m 0123456789012345678901234567890123456789012345678901234567890123456789
setenv n 0123456789012345678901234567890123456789012345678901234567890123456789
setenv o 0123456789012345678901234567890123456789012345678901234567890123456789
countenv
time execs 1000 0 nulljob
time execs 1000 1024 nulljob
time execs 1000 102400 nulljob
time execs 1000 0 bigjob
time execs 1000 1024 bigjob
time execs 1000 102400 bigjob
```