An Experimental Investigation of Scheduling Strategies for UNIX

Darwyn R. Peachey
Richard B. Bunt
Carey L. Williamson
Tim B. Brecht

Department of Computational Science
University of Saskatchewan

ABSTRACT

The scheduler used in an operating system is an important factor in the performance of the system under heavy load. This paper describes the scheduling philosophy employed in the UNIX operating system and outlines the standard scheduling strategies. Modified strategies which address deficiencies in the standard strategies are described. The effectiveness of these modified strategies is assessed by means of performance experiments.

1. Introduction

The UNIX* operating system [6] is an elegant, general-purpose timesharing system which is used on a wide range of computers. Versions of UNIX are used on computers as small as the IBM Personal Computer and as large as the IBM 3081. UNIX is available on computers built by more than fifty different manufacturers.

All true UNIX systems are derived from software developed at AT&T Bell Laboratories. Although the wide range of UNIX systems have similar user and programmer interfaces, the internals of the resident kernel are likely to be different on different computers. The UNIX software usually supplied by AT&T is designed to run on DEC PDP-11 and VAX-11 computers. Throughout this paper we will refer to the AT&T software as "standard" UNIX. In particular we will mention the versions commonly referred to as Version 6 (V6), Version 7 (V7), and System V (S5). UNIX System III is identical to System V in the areas that we discuss.

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

A timesharing system like UNIX must share the resources of the computer among multiple processes running programs on behalf of users. The allocation or scheduling of the resources has a great impact on the quality of service received by the users. In this paper we are concerned with the allocation of the CPU and main memory by the UNIX schedulers. We present a model of UNIX scheduling, and then describe the strategies used in the standard versions of UNIX in terms of this model. We suggest some deficiencies in the standard strategies and describe a set of modified scheduling strategies which we have implemented to address these deficiencies. We then describe a set of experiments which were conducted to evaluate the performance effects of the modified strategies, and discuss the results of those experiments.

2. Scheduling in UNIX

A UNIX system contains a population of processes, each of which may be *ready* to execute, actually *executing*, or *blocked* awaiting some event (usually the completion of an I/O operation). A process blocked for a long I/O operation (for example, writing to an interactive terminal) is said to be *waiting*, while a process blocked for a short I/O operation (disk I/O) is said to be *sleeping*. At a given moment, any process may be in kernel mode (running trusted kernel software with full access to the hardware), or in user mode (running user programs with a restricted instruction set and address space). Each process is represented by an entry in the system's *process table* and an *image* (user program and data), which may be either in main memory (*loaded*) or in the "swap area" on a secondary storage device. A process whose image is in the swap area is said to be *swapped out*.

Our model of UNIX scheduling is illustrated in Figure 1. As in many other operating systems, scheduling in UNIX takes place at two levels, short-term (CPU scheduling) and medium-term (main memory scheduling) [2]. CPU scheduling is done by a kernel subroutine which is called by the executing process to reallocate the CPU to another process (possibly the same process). Main memory scheduling, which determines a set of memory-resident candidates eligible for CPU scheduling, is done by swapping whole process images between main memory and the swap area. The swapping is done by a kernel process called the *swap scheduler*.

The scheduling methods employed by the CPU scheduler and the swap scheduler are often quite different, partly because rescheduling

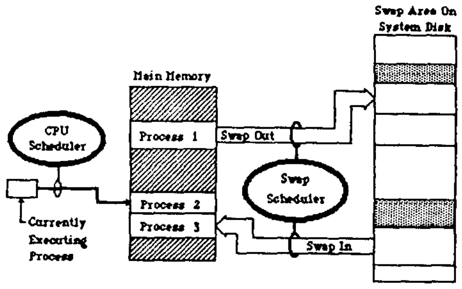

Figure 1. UNIX Scheduling Model

of the CPU occurs many times per second and must be very inexpensive, while swap scheduling occurs less frequently and may involve more expensive algorithms.

Since a process must be present in main memory before it may execute, the swap scheduler can dominate system performance if main memory is scarce. On the other hand, when memory is abundant no swapping occurs and the CPU scheduler is dominant.

Although a process may voluntarily swap itself out of memory, only the swap scheduler ever swaps a process into memory. Voluntary swapouts occur when a process is expanding and needs more than the available amount of unused main memory. By swapping itself out, the process causes the swap scheduler to allocate main memory for it when it is swapped back in.

In our comparison between the standard scheduler and our modified schedulers, we will focus on three main decisions made in scheduling:

1. The dispatching decision made by the CPU scheduler: *Which of the loaded, ready processes should be allocated the CPU?*
2. The swap-in decision made by the swap scheduler: *Which of the swapped-out, ready processes should be loaded into main memory?*
3. The swap-out decision made by the swap scheduler: *If there is not sufficient main memory available to swap in the process chosen by the swap-in decision, which eligible loaded process(es) should be swapped out to make room?* A number of rules determine which loaded processes are eligible to be swapped out. These rules are essentially the same in the standard and modified schedulers. All of the swap schedulers described in this paper distinguish between waiting processes, and sleeping or ready processes, when making the swap-out decision. Waiting processes are swapped out if possible; otherwise sleeping or ready processes are swapped out.

Our model of UNIX scheduling is designed to represent the important scheduling decisions involved in a swap-based UNIX system. Various non-standard versions of UNIX support demand-paged virtual memory, for example, the VM/UNIX systems developed by the Computer Systems Research Group of the University of California at Berkeley for the VAX computer. Scheduling strategies in such systems have some similarities to those of swapping systems, in that the CPU dispatching decision is unchanged, and swapping of processes is done to control the multiprogramming level. We have not yet attempted to deal in depth with the application of our modified schedulers to a paging-based UNIX system, but this is part of our plans for further research.

3. Standard Schedulers

The standard UNIX CPU scheduler allocates the CPU to the process with the best priority chosen from the set of loaded, ready processes. A process executing in user mode is preempted if a better priority process becomes loaded and ready. An unordered linked list of ready processes is exhaustively searched to find the loaded, ready process with the best priority. The best priority is the numerically *lowest* priority, which is unfortunately opposite to the conventional meaning of "high priority".

A priority is assigned to each process in the following way. A process in kernel mode has a priority determined by the event it last blocked for. A process in user mode has a priority determined by its *tick count*, an indicator of the amount of CPU time it has used recently. The tick count of the currently executing process is incremented every clock interrupt (16.7 milliseconds), to a maximum value of 255. Once every second, the tick count of every process is reduced by a "decay" computation that makes the tick count reflect recent CPU usage rather than total CPU usage. Various decay computations are used in standard UNIX systems: V6 decrements the tick count by 10, while V7 and S5 multiply it by 0.8 and 0.5, respectively. (The Berkeley 4.1 BSD system for the VAX uses a variant of this scheme in which the tick count is multiplied by a value between 0 and 1 depending on system load.) The priorities of processes in user mode are recomputed after the decay computation. The priority of a user mode process is computed from the tick count by means of the formula:

$$priority = \frac{tickcount}{A} + PUSER$$

where *PUSER* is the minimum (best) user mode priority, worse than all kernel mode priorities. The divisor *A* has the value 16 in V6 and V7, and 2 in S5.

In the preceding discussion we have ignored the effects of the *nice* value associated with a process. The *nice* value can be used by the system administrator and user to influence the scheduling of the process. A small *nice* value results in better service for the process. A large *nice* value results in poorer service. The *nice* value influences the rate of decay of the tick count and also directly affects the computation of the priority from the tick count.

The standard swap scheduler makes its swap-in decision by selecting the ready, swapped-out process which has been in the swap area for the longest time. The selection is implemented by means of a linear search through the entire process table. It is interesting that the priority maintained by the CPU scheduler is not used in making the swap-in decision.

The swap-out decision of the standard swap scheduler is based primarily on the amount of time that each candidate process has been in main memory. If possible, the largest loaded, waiting process is swapped out (in S5, a mixture of priority and time loaded is used instead of size). Otherwise, the ready or sleeping process that has been loaded the longest is swapped out (to prevent excessive swapping, a minimum time of 2 seconds in main memory is required). A linear search through the entire process table is used to locate the process to be swapped out.

The *nice* value is used in both the swap-in and swap-out decision algorithms to favour processes with small *nice* values.

4. Scheduler Modifications

Although the standard UNIX schedulers described in the previous section have functioned quite well in a variety of environments, there are several problems with them:

1. Since the priority of every process must change over time as its tick count decays, the system must frequently update the tick counts and recompute the priority of each process (this happens once a second in the standard schedulers).
2. The priority of a process in user mode is determined solely by its tick count, an indicator of the CPU service which it has received recently. Thus, the recognition of interactive and other I/O-bound processes is quite indirect. It takes several seconds for the tick count of a formerly CPU-bound process to decay to the level of an I/O-bound process, so sharp changes in program behaviour are recognized quite slowly.
3. The priority system is difficult to understand and tune [5]. For example, the performance effects of changing the *nice* parameter are unpredictable, because *nice* is used in different ways at several places in the CPU and swap schedulers.
4. Because the CPU scheduler is priority-based and the swap scheduler is based on the time each process has been swapped in or out, a sharp change in overall scheduling strategy occurs as the system load increases. Under a light load, no swapping occurs, and the decisions made by the CPU scheduler dominate. Under a heavy load, a great deal of swapping takes place, and the swap scheduler becomes dominant, by determining which processes are in main memory and are therefore eligible for the dispatching decision. Thus, as the load increases, the dominant strategy changes from a priority-based CPU scheduling strategy to a round-robin swap scheduling strategy that is insensitive to program behaviour.
5. Memory allocation decisions are made without any awareness of the layout of process images in main memory. This can lead to memory fragmentation problems and ineffective choices of processes to swap out.

Our scheduler modifications were motivated by these concerns. In particular, the first three problems were addressed by a new CPU scheduler. The sudden change to a round-robin strategy as system load increases was avoided by basing the swap-in decision on the same process priority information that is used by the CPU scheduler. Finally, we used a memory-oriented swap-out strategy called "MOUSE", which bases its decisions on the positions of process images and unused areas in main memory.

4.1 CPU Scheduler

Our CPU scheduler is of the FB (feedback queue) type popularized by CTSS [3] and Multics [7]. A similar scheduler is employed in VAX/VMS [4]. Each process is at one of 32 priority levels, with 0 the worst priority and 31 the best priority (Figure 2). The best priority is the highest priority, reversing the standard UNIX priority scheme in the interests of clarity. The CPU is allocated to the highest priority loaded, ready process. A process executing in user mode is preempted by a higher priority process becoming loaded and ready. The priority levels from 16 to 31 are "real-time" levels. A process with a real-time priority level is never swapped out by the swap scheduler and is allowed to execute until it blocks for I/O or is preempted by a higher priority process. The real-time priority levels are rarely used, except that the swap scheduler itself runs at level 16.

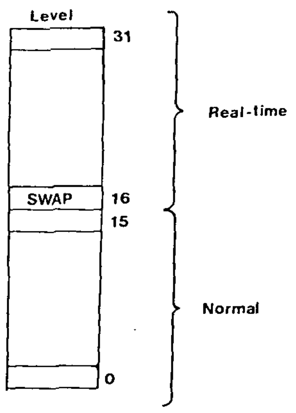

Figure 2: FB scheduler priority scheme.

Normal "timesharing" processes have priority levels from 0 to 15. Each priority level has an associated CPU quantum which is larger at the lower levels and smaller at the higher levels. Each process has a *base level* which can be adjusted to give the process better or worse service (analogous to the *nice* value in the standard CPU scheduler). When a process uses all of its quantum, its priority level is decremented, but never below the base level. When a process blocks, its priority level rises to its base level plus an increment determined by the type of event it is awaiting. The increment is 6 for terminal input, 4 for terminal output, and 2 for disk I/O. A completely CPU-bound process will have an actual priority level equal to its base level, while an I/O-bound process will have a higher priority.

The dispatching decision in the FB scheduler is much less costly than that of the standard CPU scheduler. The standard scheduler does two context switches for each dispatching decision, while the FB scheduler only does one. In addition, because the FB scheduler maintains a linked list of ready processes ordered from highest to lowest priority level, it can simply select the first loaded process in the list, rather than searching the entire list.

4.2 *Priority Swap-In Strategy*

The modified swap-in strategy simply selects the highest priority ready, swapped-out process. The desired process is easily found by picking the first swapped-out process in the CPU scheduler's ready process list. Thus, the swap scheduler attempts to swap in the swapped-out process which the FB scheduler would be most likely to select for execution if it were loaded.

4.3 MOUSE

Abdallah [1] suggested that UNIX swapping strategies should be based on the positions of process images in main memory. The MOUSE swap-out strategy implements this suggestion by maintaining a complete map of main memory as memory areas are allocated and released (standard UNIX uses a map that indicates only unused memory). Each map entry indicates the size of the associated area, the address where it begins, and whether it is used or unused. In the case of a used area, the map entry also includes a pointer to the process table entry for the process image occupying the area. The map entries are stored in an array ordered by the addresses of the corresponding memory areas.

The MOUSE swap-out strategy selects processes to swap out by searching through the memory map from low addresses to high addresses. MOUSE looks for the first adjacent cluster of swappable process images and unused areas which is large enough to accommodate the incoming process. Because some processes cannot be swapped out, it is not guaranteed that such a cluster exists. MOUSE makes an initial pass through the map considering only waiting processes to be swappable. If a large enough cluster cannot be found under this constraint, MOUSE makes a second pass considering waiting, sleeping, and ready processes. When a large enough cluster is found, MOUSE proceeds to swap out the processes in that cluster.

One advantage of the MOUSE swap-out strategy is that no processes are swapped out if a large enough cluster cannot be found. In such a case, the standard swap scheduler would swap out processes even though doing this would not result in a large enough unused area for the incoming process. By eliminating these ineffective swap-outs, CPU and memory utilization are improved.

5. Design of the Experiments

A series of experiments was conducted to measure the performance effects of the various scheduling strategies described above. All of the experiments used the same version of the UNIX kernel, running on a DEC PDP-11/23 minicomputer in our Department's Research Laboratory. This version of the UNIX kernel has been instrumented to record more than thirty different types of internal kernel events. Five different scheduling modules were "plugged into" the kernel to systematically vary the scheduling strategies, as follows:

1. The standard V7 scheduling module, used as a baseline for comparing the modified schedulers to the standard ones.
2. FB/std/std, a scheduling module using the FB CPU scheduler and the standard swap-in and swap-out strategies. Since FB/std/std uses the standard swap scheduler, any performance differences between it and the V7 scheduler are attributable to the FB CPU scheduler.
3. FB/pri/std, which uses the priority-based swap-in strategy and the standard swap-out strategy.
4. FB/std/mouse, which uses the standard swap-in strategy, and the MOUSE swap-out strategy.
5. FB/pri/mouse, which uses all three of our modified scheduling strategies.

Each of the five scheduling modules was tested in two different experiments, one to measure system throughput, and one to measure the degradation in the response time of an interactive process as system load increased.

Each throughput experiment consisted of eight separate test runs. Each run involved the completion of a synthetic workload of CPU and disk I/O activity consisting of 36 invocations of a "load" subroutine. The workload was designed to be similar to the load produced by the common utility programs (*e.g.*, the C compiler, and the text formatters) that account for most CPU and disk usage in a typical UNIX environment. The real time used to complete the workload was measured and used to compute overall throughput in "loads per minute". At the same time, all available internal monitoring data was recorded, including the number of swap operations, the CPU utilization, and the number of dispatches and system calls. The workload was spread over a different number of processes in each of the eight runs (1, 2, 3, 4, 6, 9, 12, and 18 processes, respectively). Despite the varying number of processes, the same total amount of work was done by each run (this is supported by the fact that every run executed the same number of system calls). The sizes of the different processes in a given run were varied so that no unrealistic memory allocation tricks were possible.

The goal of the experiments, as mentioned earlier, was to compare the performance impact of the different scheduling strategies in the context of UNIX. To ensure that differences in the results of the experiments were attributable to the different scheduling strategies, it was necessary to limit variations in the results from other causes. The following precautions were taken to reduce such variations:

1. Use of the same synthetic workload in all runs eliminated variations due to load. Each experiment was performed with the computer idle except for the experiment.
2. Before each run, a set of large CPU-bound processes was used to force inactive processes out of memory. After the CPU-bound processes terminated, memory was in a known state (empty except for the shell process supervising the experiments).
3. To eliminate variations due to the placement of blocks on the file system and the effects of UNIX's disk cache, disk I/O was performed on a dedicated disk drive, and bypassed the file system entirely. The pattern of physical disk I/O produced by the synthetic workload simulated the pattern that would be produced by file system accesses in normal operation.

With these precautions, the results of several runs of the same experiment were within one percent of each other. Thus the larger differences observed in the results of different experiments can reasonably be attributed to the effects of the different scheduling strategies.

The response degradation experiment involved nine separate runs for each of the five systems tested. In each run the completion time of an interactive test process was measured, while a background load of large, CPU-bound processes was also applied to the system. The nine runs used 0, 1, 2, 3, 4, 6, 9, 12, and 18 background processes, respectively. The interactive process simply performed 20 read operations on the console terminal. UNIX alarm signals were used to ensure that each read operation was forced to wait 10 seconds before finishing. Thus, with no other system load, the interactive process finished the 20 reads in 200 seconds. With a large number of background processes running, the interactive process could take much longer to complete because of the delay between the time that each read operation completed and the time that the process was loaded into main memory and executed so that it could start the next read operation.

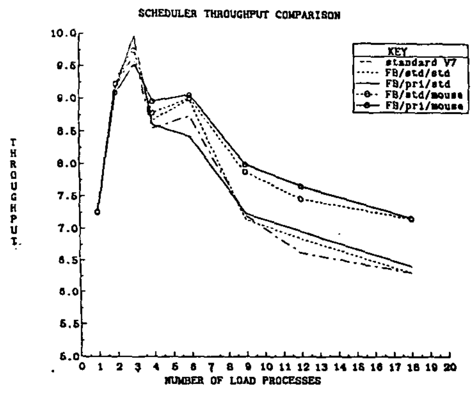

Figure 3: Scheduler throughput comparison.

6. Results of the Experiments

The results of the experiments are shown in a series of graphs in Figures 3 through 7. Each graph shows five lines representing the five different scheduling modules which were tried.

The first four graphs show results from the throughput benchmark: the measured throughput, the number of swap-out operations, the percentage user-mode CPU utilization, and the number of dispatches. Throughput is shown as a rate, in loads per minute, where a load is a fixed amount of CPU and disk I/O work as described in the preceding section. The swap-out and dispatching comparisons show the total number of these operations which took place during each experiment. It is meaningful to compare these numbers, since all experiments performed a total of 36 loads even though the number of processes was varied.

The throughput graph in Figure 3 can be analyzed in two parts. When very few processes are active, the differences between the systems are small and difficult to interpret. With fewer than four processes, almost no swapping takes place, so performance differences are due to CPU scheduling and random variations. The increase in throughput up to this point is a result of the system overlapping user CPU activity and disk I/O. Figure 5 shows that three processes gives maximum user-mode CPU utilization as well as maximum throughput.

When a large number of processes are active, the situation is quite different. Swapping decisions become the dominant factor in throughput. From Figure 3 it is clear that the five systems can be divided into two groups based on their throughput when a large number of processes are active. The two systems with the MOUSE swap-out strategy have approximately ten percent higher throughput than the three systems with the standard swap-out strategy (in the 18 process case). The priority-based swap-in strategy seems to result in slightly better throughput under heavy load, but has a much smaller impact than the MOUSE swap-out strategy.

The swap-out comparison in Figure 4 indicates that the throughput advantage of the MOUSE strategy is largely a result of reduced swapping. This is not surprising, since MOUSE attempts to make better use of main memory and more effective choices of processes to swap out. Since swapping consumes CPU and disk resources and does not contribute to the productive work done, a high swap rate definitely has an adverse effect on system throughput. The experimental results clearly demonstrate this inverse relationship between swapping and throughput.

The user-mode CPU utilization graph in Figure 5 indicates that the MOUSE swap-out strategy also results in better utilization of the CPU by user programs. This is partly a result of reduced

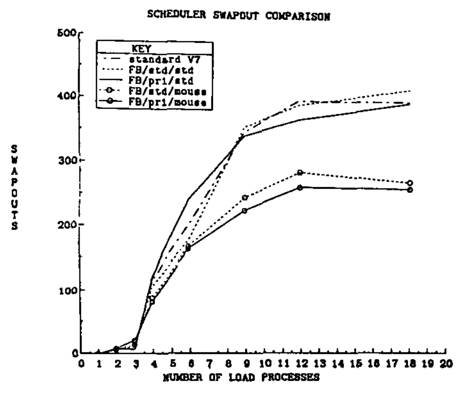

Figure 4: Scheduler swapout comparison.

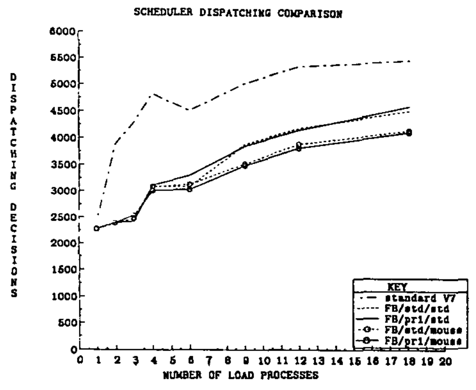

Figure 6: Scheduler dispatching comparison.

system overhead from swapping. However, there is also a significant reduction in the percentage of time that the CPU is wasted because all ready processes are swapped-out. MOUSE makes better use of main memory, and therefore is able to keep a somewhat higher number of processes in memory. Internal measurements show that this effect is about three times as significant as the reduction in system overhead.

Figure 6 demonstrates the largest performance advantage of the FB scheduler. The number of dispatches performed by the V7 system with the standard CPU scheduler is markedly higher than the number of dispatches performed by the other systems. The difference is large beginning at the 2 process case and extending to the 18 process case. Since the cost of a dispatching decision is more than twice as great in the standard CPU scheduler as in the FB scheduler, this is an important reduction in kernel overhead. Because of somewhat lower swap scheduler activity, the systems with MOUSE performed a smaller number of dispatches than the systems with the standard swap-out strategy when a large number of processes were active.

The final graph (Figure 7) shows the results of the interactive response experiment. The interactive test process completes in the minimum time of 200 seconds with a small number of CPU-bound background processes running.. However, once the number of active processes forces swapping to occur, all three systems with standard swap-in strategies take considerably longer to run the interactive process. By contrast, both systems with the priority swap-in strategy continue to run the interactive test process in 200 seconds even with 18 CPU-bound background processes running. Clearly the swap-out strategy has no significant effect in this experiment. The FB scheduler is important in that it provides the quick

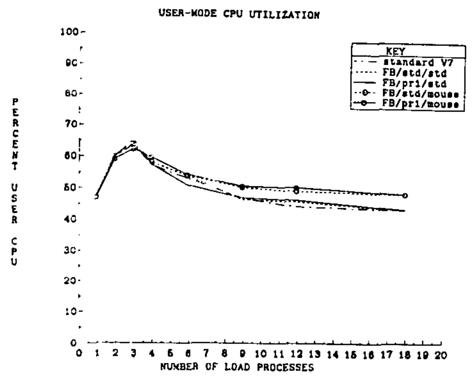

Figure 5: User-mode CPU utilization.

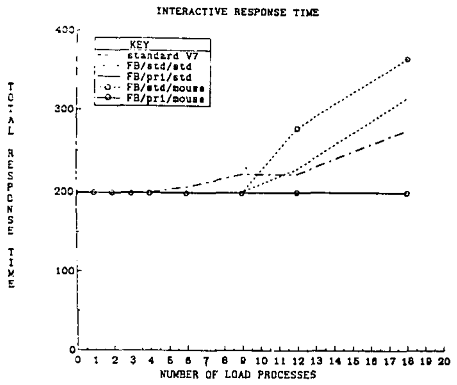

Figure 7: Interactive response time.

recognition of interactive processes which is used by the priority swap-in strategy.

Although the CPU efficiency of the swap scheduling algorithms was not our primary concern, we should note that the priority-driven swap-in strategy and the MOUSE swap-out strategy are both somewhat less costly than the standard strategies. The standard scheduler makes its decisions by means of linear searches through the entire process table; our algorithms search much smaller data structures, namely a linked list of the ready processes for the swap-in decision, and the memory allocation map for the swap-out decision.

7. Conclusions

Scheduling is an important factor in determining the behaviour of an operating system under heavy load. This paper has examined the three key decisions made by UNIX schedulers. A set of experiments was conducted to study the performance effects of various alternative strategies for making these decisions.

The experimental results indicate that a priority-based swap-in strategy is very effective in maintaining the responsiveness of interactive processes on a busy, memory-limited UNIX system. MOUSE, a swap-out strategy based on memory management concerns, is also quite effective at improving the throughput of such a system, by reducing the number of swap operations performed. The FB CPU scheduler results in reduced dispatching overhead and provides the quick recognition of interactive processes necessary for the priority swap-in strategy.

8. Acknowledgements

We gratefully acknowledge the assistance of Malak Abdallah and David Bocking. Malak's M.Sc. thesis proposed an earlier version of the MOUSE scheduler, while Dave's continuing technical support as Manager of the Department's Research Laboratory has been most valuable. The comments of the anonymous referees contributed to the clarity of the final paper. The research was funded, in part, by the Natural Sciences and Engineering Research Council of Canada, through operating grant no. A3707, infra-structure grant no. A2527, and the Undergraduate Summer Research Awards program.

9. References

[1] Abdallah, M.S., *An Investigation of the Swapping Process in the UNIX Operating System*, M.Sc. thesis, Department of Computational Science, University of Saskatchewan, July 1982.

[2] Bunt, R.B., "Scheduling Techniques in Operating Systems", *Computer*, Vol. 9, No. 10 (October 1976), 10-17.

[3] Crisman, P.A., ed., *The Compatible Time-Sharing System*, MIT Press, 1965.

[4] Digital Equipment Corporation, *VAX Software Handbook*, 1982.

[5] Peachey, D.R., Williamson, C.L., and Bunt, R.B., "Taming the UNIX Scheduler", Dept. of Computational Science, Univ. of Saskatchewan, 1984, (submitted for publication).

[6] Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System", *Comm. ACM*, Vol. 17, No. 7 (July 1974), 365-375.

[7] Saltzer, J.H., *Traffic Control in a Multiplexed Computer System*, Sc.D. thesis, Department of Electrical Engineering, MIT, 1966.