Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits †

Özalp Babaoğlu ‡

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California 94720

Abstract- This paper discusses the modifications made to the UNIX operating system for the VAX-11/780 to convert it from a swap-based segmented system to a paging-based virtual memory system. Of particular interest is that the host machine architecture does not include page-referenced bits. We discuss considerations in the design of page-replacement and load-control policies for such an architecture, and outline current work in modeling the policies employed by the system. We describe our experience with the chosen algorithms based on benchmark-driven studies and production system use.

Beginnings

In the fall of 1978 the Computer Science Division of the University of California at Berkeley purchased a VAX-11/780* and arranged to run an early version of UNIX** for the VAX provided by Bell Laboratories under a cooperative research agreement. The VAX was purchased because it is a 32-bit machine with a large address space, and we had hopes of running UNIX, which was being used on other smaller machines.

† This material is based upon work partially supported by the National Science Foundation under Grants No. MCS 7807291, MCS 7824618, MCS 7407644-A03, Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4031, Monitored by the Naval Electronics Systems Command under Contract No. N00039-80-K-0649 and by an IBM Graduate Fellowship to the second author.
‡ Author's present address: Department of Computer Science, Cornell University, Ithaca, New York 14853.
* VAX and VMS are Trademarks of the Digital Equipment Corporation.
** UNIX is a Trademark of Bell Laboratories.

Except for the machine-dependent sections of code, UNIX for the VAX was quite similar to that for the PDP-11 which has a 16-bit address space and no paging hardware. It made no use of the memory-management hardware available on the VAX aside from simulating the PDP-11 segment registers with VAX page table entries. The main-memory management schemes employed by this first version of the system were identical to their PDP-11 counterparts--processes were allocated contiguous blocks of real memory on a first-fit basis and were swapped in their entirety. A subsequent version of the system was capable of loading processes into noncontiguous real memory locations, called _scatter loading_, and was able to swap only portions of a process, called _partial swapping_, as deemed necessary by the memory contention. This would become the basis for the paging system development discussed in this paper.

Goals

The user-friendliness and portability of the UNIX environment were perceived to be large advantages by our installation. However, application programs outgrew our resources with the original swap-based system very quickly. The initial configuration of the machine had only 1/2 megabyte of real memory. Although, in the long run more memory would be available, it seemed natural to incorporate paging into UNIX and thereby support larger applications and make better use of our limited main memory. This would also provide a vehicle for researching storage hierarchy performance evaluation and memory-management techniques.

Search for a replacement policy

The VAX memory-management architecture supports paging within three segments (two for user processes, one for the system). The interesting aspect of the architecture is the lack of page-referenced bits (also called _use_ bits). Such bits typically provide the reference information which commonly-implemented page replacement algorithms such as ''clock'' and Sampled Working Set (SWS) [DENN 68a] (to be described in the following sections) base their decisions on. Without even this minimal page reference information, the only reasonable algorithms for replacing pages are the First-In-First-Out (FIFO) and the Random (RAND) policies, which are known to have performances (as measured by the number of page faults generated for a given mean memory occupancy) inferior to the clock and SWS policies [BELA 66, KING 71]. To remedy this situation, the dynamic address translation mechanism of the VAX was used to detect and record references to pages. With this scheme, a page for which reference information is to be gathered is marked as invalid although it remains in main memory. This state for a page is called the _reclaimable_ state. A reference generated to a location within this page causes an _address-translation-not-valid_ fault. However, the fault handler can detect this special state of the page and thus refrains from initiating the page transfer from secondary memory. In other words, the reclaimable state for a page corresponds to a valid page with the reference bit off if the reference bit were available. Since this method of simulating page-referenced bits through software has a nonnegligible cost associated with it, the relative performance of some of the well-known replacement algorithms in this environment is no longer obvious.

In VMS, the vendor-supplied operating system for the VAX, the solution to the replacement decision is simple. Each process is assigned a fixed-size memory partition, called a _resident set_, that is managed according to the FIFO policy. Pages that are not members of any of these resident sets are grouped together to constitute the global free list which functions as a disk cache. Although there is some isolation between the paging behavior of the various processes due to the strictly local resident sets, the coupling that is introduced through this global free list has significant performance implications. Lazowska [LAZO 79] reports that in his measurements based on a real workload, system performance was significantly improved by increasing the minimum size of the free list (a system generation parameter). An unfortunate consequence of allocating fixed-size partitions to processes is that a process has its pages taken away from its resident set (relatively small in size compared to the total real memory available on the machine) and placed in the free list to be subsequently reclaimed even though it may be the only active process in the system.

Babaoglu has studied a class of _hybrid_ replacement policies that employ different algorithms for page replacement amongst two logical partitions of pages in main memory [BABA 80]. This class includes the VMS algorithm described above as an instance where the resident set management is according to the FIFO policy and the free list management is approximately Least-Recently-Used (LRU). [BABA 80] shows that for a given program and a given amount of available memory, there exists a resident set size for which the FIFO-LRU hybrid policy achieves a fault rate close to that of the pure LRU policy while incurring a cost comparable to that of the FIFO policy.

UNIX is particularly ill-suited for such a scheme for several reasons. The UNIX system encourages the creation of a number of processes to accomplish most tasks-- processes are cheap. These processes are nonhomogeneous; they vary greatly in size and in the manner in which they access their address space. Furthermore, in certain processes the page reference behavior varies radically over time as the process enters different phases of execution. The LISP system, which initiates garbage collection after an interval of execution, is an example of such a process. Thus, in this environment, it is unlikely that we will find a single system-wide value for the fixed resident set size that will nearly optimize a cost function that is the weighted sum of the page fault rate and the rate at which reclaimable pages are referenced for the hybrid policy. In fact, even for a single process, the value of the resident set size must vary in time in order to track different phases of its execution and the varying amounts of real memory available to it. As described earlier, the total number of pages from the free list belonging to a certain process is a dynamic quantity due to its sensitivity to the system-wide paging activity. A more recent version of the VMS operating system (version 2.1) attempts to remedy some of these problems by adjusting the process resident set size within two fixed boundaries according to a heuristic based on global paging rates [DEC 80]. Due to its unavailability at the time, this modified version of the system was not included in our studies.

Simulation studies based on actual program address traces showed the clock page replacement algorithm [CORB 68] to be much more robust with respect to the cost function defined above to variations in the amount of memory available to the program, the relative costs of page faults and reclaims, and the nature of the program itself than the fixed-partition VMS scheme [BABA 81a]. Under the simplest form of this policy, all the pages allocated to a program are thought of as ordered around the circumference of a circle, called the _loop_, according to their physical page frame number. In addition, there is pointer, called the _hand_, that is advanced circularly through them when page faults occur until a replacement candidate is located. A page is chosen for replacement if it has not been referenced during the time interval between two successive passages of the hand through this page. Empirically, the clock page replacement policy achieves fault rates that are very close to those of the LRU policy although it is much easier to implement [GRIT 75]. On machines with reference bits, it suffices to examine reference bits associated with pages as the hand passes over them. If a page has the reference bit clear when the hand passes, it has not been referenced for one revolution and thus it is selected for replacement. If a page has been referenced, then the reference bit is cleared and the page remains in the loop for at least another revolution. This examining of the reference bit along with the associated action is called the _scan_ operation. For our environment, this algorithm can remain unchanged since setting the reference bit associated with a page corresponds to moving it from the reclaimable to the valid state whereas resetting its reference bit corresponds to moving it from the valid to the reclaimable state.

Another major departure in our UNIX memory management from the VMS design resulted from our decision to apply the clock page replacement algorithm globally to all pages in the system rather than locally to the pages for each process. This results in a variable-size memory partition for each process. This was motivated by studies where global versions of fixed-partition replacement policies had been found to have better performances than their local counterparts [OLIV 74, SMIT 80, SMIT 81], and some special properties of our environment,

(i) The relative simplicity of the global clock policy and, consequently, the ease of implementation.

(ii) The projected workload for the system had no requirement of guaranteed response times as in real-time applications.

(iii) It was unreasonable to expect users to specify the sizes of the fixed program partitions since from the existing system they had little or no information about the memory requirements of programs.

(iv) Without reference bits, the cost of implementing variable-partition local replacement policies such as SWS and Page Fault Frequency [CHU 76] was too high. We further comment on this in the following section.

(v) UNIX encourages the construction of tasks consisting of two or more processes communicating through *pipes*, which must be co-scheduled if they are to execute efficiently. In most instances, the activity intensity, thus the memory demand, shifts over time from the left-most process to the right-most process in the pipe while all of them remain active. It was our belief that in such an environment, dynamic partitioning of memory amongst these processes in real time is more appropriate than having local partitions (working sets) that are maintained in process virtual time.

Memory demand and clock triggering

The clock page replacement policy is only engaged upon a page fault, at which time it selects a page to be replaced. Given that the demand for memory exhibits nonuniform patterns with occasional high spikes (see Figure 1), this strategy for the activation of the replacement policy is clearly suboptimal.

Having incurred the cost of page replacement policy activation, we would like to select more than a single page to be replaced in order to anticipate short-term demand for more memory. To this end, the system maintains a free page pool containing all of the page frames that are currently not in the loop. Our version of the clock policy is triggered whenever the size of this pool drops below a threshold. Then, the algorithm scans a given number of pages per second of real time (a simplified version of this algorithm is discused in [EAST 79]). Currently, the default trigger point for the free page pool size is set at 1/4 of the real memory size and the default minimum scan rate of the hand is approximately 100 pages per second. As the free page pool size further drops below the threshold, the scan rate of the hand is increased linearly up to a given maximum value. The primary factor that determines this maximum value is the time that it takes to service a page reclaim from the loop (i.e., the time to simulate the setting of a reference bit). Measurements based on the current system indicate that this action consumes approximately 250 microseconds of processor time. Since the number of pages scanned by the clock algorithm provides an upper bound on the number of pages that can be reclaimed, the processor overhead due to the simulation of reference bits can be controlled by limiting this maximum scan rate. Currently, we allocate at most 10 percent of the available processor cycles to this function which implies that the maximum scan rate of the hand is limited to approximately 300 pages per second. Due to the existence of the free page pool, however, short duration memory demands far in excess of this value can be satisfied.

The system maintains enough data to be able to reclaim any page from the free page pool regardless of how it arrived there. In addition to being replenished from the loop, the free page pool also receives pages of processes that are swapped out or completed. In both cases, these pages can be reclaimed by the process upon a subsequent swap in or a future incarnation of the same code, provided of course that the pages have not been allocated for another purpose.

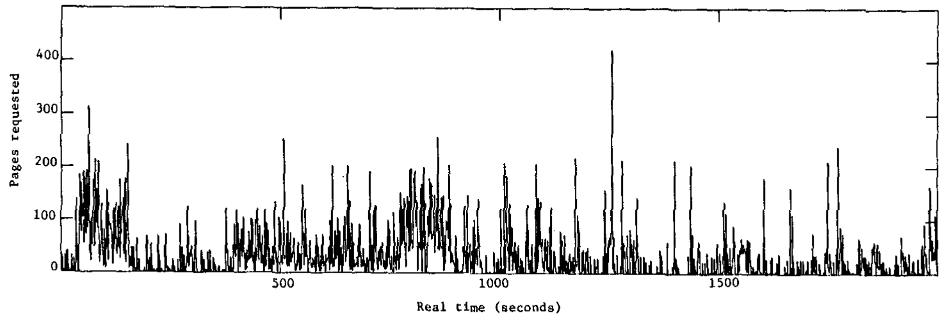

Figure 1. Number of page frames requested globally in one second intervals during a 33 minute observation period.

Given the cost to simulate the setting of a reference bit, our previous remark concerning the unsuitability of local variable partition page replacement policies in the UNIX environment is justified. As an example, using the Sampled Working Set policy with a window size of 100,000 instructions (approximately 100 milliseconds on the VAX) operating with a program having a 400-page working set would consume 100 percent of the processor cycles just to simulate reference bits (assuming that the working set of the program remained unchanged between two consecutive sample points).

The use of a modified clock page replacement algorithm where the scan rate is based on the available memory has several other advantages, as well. The length of the free page pool becomes a natural indicator of the amount of memory contention in the system. As we shall see, the inability of the system to maintain some specified amount of free memory is the basis for load control, and causes process deactivation by swapping. Control of the rate of the scan allows modified page write-back activity, that is initiated when dirty pages are removed from the clock loop to be spread more uniformly over time, thereby easing contention for disk.

Implementation; new system facilities

The UNIX system memory-management facilities are particularly simple. Each user process has a read-only shared program area, a modifiable data area, and a stack. An exec system call overlays a process' address space with a particular program image from a file consisting of the shared code and the initialized data. New processes are created by the fork system call which causes a process to duplicate itself. Usually, the command interpreter accomplishes its task by first creating a copy of itself to establish the context for the command and then causes this copy overlay itself with the file that is the image of the command. Except for shared program areas, no shared memory between processes is available. Access to files and devices is through read and write system calls; no segment-based or page-based shared access to file pages is available.

Consistent with our design goals, we wished to keep changes to the system as simple as possible and orthogonal to the rest of the system design. Then, further changes in the UNIX system would not invalidate our efforts.

The conversion of the swap-based system to a paged system began in the late spring of 1979 and the first version of the paging system was put into production use on a single machine in September of 1979. At that time, the primitives for the swap-based UNIX system were still in use. Processes were created using the fork system call which copied a process' address space page-by-page to create the new address space. This newly-formed address space was then overlaid with a new image through the exec system call. These primitives, while simple to implement and relatively cheap (involving memory-to-memory copy and file reading) in a swap-based system, were very expensive under the new system, since programs might be partially loaded in memory and could be much larger.

We found that a vast majority (over 80 percent) of all forks executed in the system were due to the command interpreter. Since these forks only serve to establish the context for the new process, duplication of the entire address space was wasted effort. Most of the sharp spikes in the global memory demand pattern of Figure 1 could be attributed to processes forking and/or execing. The nondemand nature of these requests for memory (in the sense that they are an implementation artifact) overtaxed the page replacement algorithm and had grave performance consequences.

A natural solution to the problem would have been to include a ''copy-on-write'' facility to implement a fork similar to that used in various PDP-10 operating systems (such as TENEX [BOBR 72]). In this scheme, the two processes would be allowed to share the same address space and the copying at the page level would be deferred until the time of the first modification of a page by either process. However, this would have significantly increased the number of modifications to UNIX and hence delayed the completion of a workable and useful system. At the time, the desires of our user community did not indicate that shared-memory primitives would be necessary in the near future. Copy-on-write paging seemed to introduce a good deal of complexity into the relatively simple system data structures to warrant support for the very small amount of computation which occurs between a fork and an exec system call.

A new primitive to replace most instances of the fork system call was designed. This primitive, called virtual-fork, allows the original process to establish the system context for the new process but refrains from creating the address space until the subsequent exec system call that is issued by the new process or the completion of the new process. During this interval, the system context of the original process is dormant. To put it another way, the new process is allowed to run within the address space of the original process until it establishes its own address space through an exec system call or completion at which point the original process, which was dormant, regains its address space. Obviously, during this transition time, the new process must not modify the contents of the address space that is ''on loan'' to it. This mechanism allows a new process to be created without any copying of address space and without requiring a mechanism like ''copy-on-write.''

Note that there are instances of process creation where the virtual-fork system call is inappropriate. An example of such a case occurs when commands are executed in the ''background.'' Then, the new process is initiated but the command interpreter does not wait for its completion and is ready to accept a new command line. However, all other instances of the fork system call could be (and were) replaced with the virtual-fork call without change to the calling program. It is quite easy to implement this primitive on non-paged machines as well as paged machines, and there are strong indications that the overhead of process creation in the swap-based PDP-11 implementation of UNIX would be reduced if such a primitive were implemented.

A new load format was also provided to reduce the implied overhead of the exec call. Programs loaded using this new format would have their pages demand-loaded from the file system rather than pre-loaded as in the previous swap-based system. This reduced the overhead of process invocation, and was soon made the default load format.

Limiting page traffic and controlling multiprogramming load

In addition to the processor overhead considerations which limit the scan rate of the clock replacement algorithm, there are global system considerations involved in limiting page traffic. Input-output activity generated by page replacement should not displace too much of the input-output activity generated by program request. UNIX typically runs on relatively small machines that usually have only two moving head disk drives which are used for all system activity including paging, swapping and file system transfers. Special paging devices are rare in such systems. It is not practical to design a system that saturates one of these arms to maximize memory usage. Input-output bandwidth is often as precious as memory residency. Load control mechanisms such as the ''L=S'' or the ''50 percent'' criterion [DENN 76, DENN 77], which assume the availability of a separate paging device, are therefore inappropriate. We therefore decided to deactivate processes by swapping them to secondary storage when demand for main memory exceeded our ability to supply it.

Multiprogramming load control in our system is thus based on a desire to limit paging overhead. When the system finds that it cannot maintain an acceptable amount of free memory while consuming approximately 10 percent of the available processor time to sample page utilization it lowers memory demand by removing a process from the set of runable processes. The process to be swapped out is selected by choosing the oldest amongst the n largest resident processes. This policy represents a compromise between the largest-first and the oldest-first policies [COFF 73]. Neither of these policies was found to be satisfactory in its pure form; the former prohibits a large process from making any progress while the latter wastes effort by constantly swapping out small processes that do not contribute much to the memory demand. Currently, the default value for the variable n is 4. The pages of the swapped-out process are written to secondary storage if necessary, and removed from the loop and returned to the free list. Processes that are swapped out are assigned priorities to return to the runable set based on their size (smaller jobs have higher priority) and the amount of time they have been swapped out (priority increases as time goes by). Sufficient time delay is built into the swapping algorithm to ensure that useful work gets done between swaps. Since in a reasonably-configured system swapping out a process is a rare event, we do not swap in the resident set a process had at the time it is swapped out. In our environment, the long period of inactivity that caused the swap out is usually a leading indicator of a locality transition through the invocation of a new function (for example, a new input line to the command interpreter). In such cases, the overlap between the old resident set and the new is minimal. However, even with an initially empty resident set, chances are the process will find some of its pages in the free page pool, and can simply reclaim them by referencing them.

Modeling the free page pool

The purpose of the free page pool is to ''smooth'' the high frequency components of the memory demand by absorbing the sharp peaks with little resistance. To accomplish this, the free page pool requires periodic replenishment. Page replacement, process completion and process swap out replenish this pool, the latter two without explicit action by the page replacement policy. Our variant of the ''triggered sweep'' clock page replacement algorithm with varying sweep rates has three parameters: the free memory threshold at which it is engaged, the scanning rate at this threshold, and the maximum scanning rate. Unfortunately, this does not lend itself easily to analytic modeling efforts.

To formalize the free memory control policy, one can view the free page pool as a stock room containing a certain inventory of a commodity and memory requests as the demands for that commodity. We can then apply inventory control theory to our problem with the hope of selecting a policy for the replenishment action (the ''order point'' and the ''order quantity'' in inventory control terminology) that can be demonstrated to be stochastically optimal with respect to a certain objective function defined for the process. Using a mapping of the costs involved in the classical inventory problem to the problem at hand and an adequate modeling of the stochastic demand process, we can obtain policy parameters that will result in approximately minimum cost in the long run. This effort is currently underway and will be reported in [BABA 81b].

Comparison with the swap-based system

After two months of production use and a reasonable amount of tuning, we decided to compare the performance of the system running with and without the virtual-memory changes. A script-driven experiment was designed for a stand-alone configuration consisting of 1 megabyte of main memory, two disk arms on two different controllers, each with a peak transfer rate of 1 megabyte per second and a 40 millisecond average access time. For the comparison we used the version of the swap-based system that was the base for the paging developments. The page size in use in the paging version of the system was 512 bytes.

The basic unit of work generated by the script was made up of four concurrent terminal sessions:

lisp A LISP compilation of a portion of the LISP compiler, followed by a ''dumplisp'' using the lisp interpreter to create a new binary version of the compiler.

ccomp An editor session followed by the compilation and loading of several small programs that support the line-printer spooler.

typeset An editor session followed by the typesetting of a mathematical paper and production of output for a raster plotter.

trivial Repeated execution of a trivial command (printing the date) every few seconds.

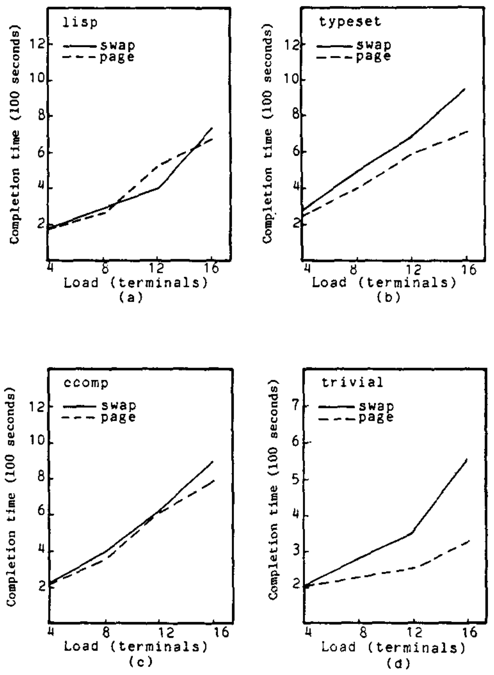

Figure 2. Average completion times
(a) lisp, (b) typeset, (c) ccomp, (d) trivial.

Staggered multiple initiations of from one to four of these terminal sessions were used to create increasing levels of load on the system. Figure 2. gives the average completion times for each type of session under the two systems. For the nontrivial sessions, completion times were very similar under the two systems, with the paging version of the system running (in all but one case) faster. The interesting observation is that the swap-based system departed from linear degradation more rapidly. This trend is most noticeable in the response time for the trivial sessions.

Figure 3 gives system-wide measurements collected during the same experiments whose results were given in Figure 2. These measurements show the same trend for both the time when the last script completed execution and average completion times for individual sessions, with the paging system slightly faster and degrading more linearly than the swap system within the measured range. Under heavy load, system overhead was uniformly greater under the paging system, constituting 26 percent of the CPU utilization as compared to 20 percent under the swap system. User-CPU utilization under this load was, however, uniformly greater for the paging system, averaging 48 percent, while the swap-based system averaged only 42 percent.

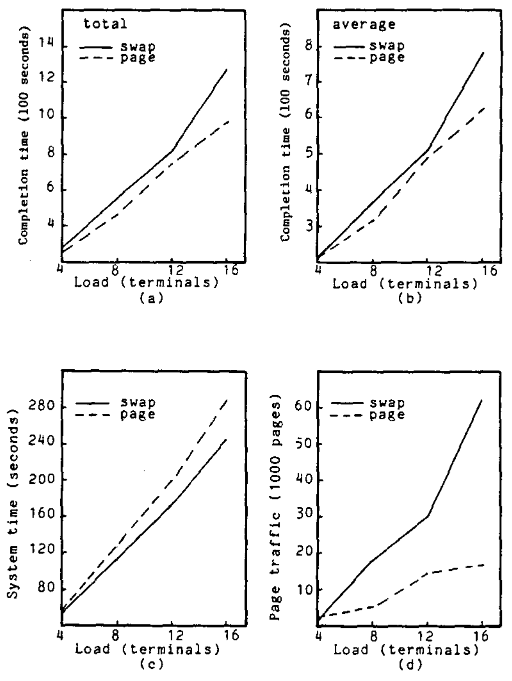

Figure 3. System-wide measurements.
(a) total completion time, (b) average completion time, (c) system time, (d) page traffic.

Finally, the total page traffic generated under the two systems was measured. The measurement accounts for both paging and swapping traffic under the paging system, as well as transfer of all system information (control blocks and page tables) under both systems. Although the paging system resulted in far fewer total pages transferred, the number of transactions required to accomplish this was much greater since most transfers under the paging system were due to paging activity rather than swapping activity. In this version of the paging system, all paging input/output activity was on single 512 byte pages.

Performance enhancements and comparisons with hybrid paging

After measuring the system and seeing that the performance was comparable with the swap system, we determined that there was a major bottleneck in the system due to the small page and file block size-- 512 bytes. Measurements of typical system programs which processed files one character at a time showed that the fastest such programs produced and consumed data at a rate of about 80 512-byte pages per second. The file system in use on UNIX at that time, however, could produce about 40 blocks per second on average, resulting in a factor of two mismatch between typical program speed and average file system throughput.

The file block size was increased from 512 to 1024 bytes and physically adjacent pages were grouped in pairs producing the current 1024-byte ''pages''. In this paper, all future references to ''pages'' will imply this new size unless noted otherwise. With the new page and file block size, total system throughput on the script-driven benchmarks discussed above improved significantly, with the completion time dropping an average of 30 percent, user-CPU utilization rising nearly 20 percent and system overhead dropping below that of the swap-based system.

Benchmarks of paging intensive synthetic programs run on VMS and UNIX showed, however, that UNIX could not supply memory to heavily paging programs at a rate comparable to VMS [KASH 80]. Simple test programs that sequentially or randomly (with varying degrees of randomness) accessed virtual memory were run on both systems and ran much faster on the VMS system which clustered pages both on input and output. The problem, here, was similar to the problem with the file system: inadequate blocking. Transferring only 1024 bytes of data after incurring 25-30 milliseconds while waiting for a moving-head storage device kept the bandwidth low.

To remedy the situation, a simple form of pre-paging was implemented. Upon a page fault, the faulting page as well as the next several virtually (and physically) adjacent pages were read in as a single operation. Similarly, upon a page out decision, the set of modified pages would be searched to construct clusters of virtually (again also physically) adjacent pages that would be cleaned in a single operation. Both the input and output cluster sizes are variables that can be varied while the system is in operation. This drastically improved system performance on the simple test programs due to their sequential nature and the fact that they always dirtied pages by writing into them.

There remained, however, a performance gap between our system and VMS whose cause eluded us at the time. The problem was discovered to be the placement of pre-paged data. Such data was placed in the clock loop, but marked as being not referenced, so it would be moved to the free page pool in a single revolution of the clock if it remained unreferenced by the program. For programs like the test program, which have a very high data rate but do not use all the prefetched data, this resulted in an excessive load on the clock algorithm.

This flaw in the pre-paging algorithm was corrected by placing the pre-paged pages at the bottom of the free page pool list rather than the clock loop. Recall that the system free page pool, which is implemented as a queue, is fairly long. On a busy system, pages near the bottom of this list may survive (i.e., remain reclaimable) for a few seconds before being re-used. Since the pages were pre-paged because they were adjacent to a recently referenced page, it is desirable to retain them only for a short while if they are not referenced. The modified pre-paging placement policy more closely reflects this intent.

A new system call was added to notify the system that a process would be exhibiting anomalous behavior. This call caused the reference bit simulation to be turned off resulting in approximately random page replacement (since the physical ordering of page frames in the free page pool from where they are allocated is destined to be random after a period of operation of the system) for these processes. Currently, the LISP system issues such a call before entering the garbage collection phase.

After these changes, the performance of the two systems on the test programs became comparable. In practice, however, the UNIX page replacement algorithm has the advantage that it does not give processes fixed partitions and therefore tends to avoid unnecessary processor overhead (a different form of thrashing [DENN 68b] that is unique to our environment) in a way that a fixed partition scheme cannot do. We are currently measuring the performance of different pre-paging and clustering strategies using trace data that was collected from the system. It is hoped that we can develop a model for the different techniques and justify or improve on the current system algorithms. [JOY 80] gives more information on the current performance characteristics of the UNIX system on the VAX.

User experience and future directions

Even before the performance improvements described above were incorporated, the system had met its original goals by being able to support applications that we could not earlier. Distribution of the system to other VAX UNIX sites began in January 1980, and over 50 other sites were running the system in the spring of 1980. The decision to keep the system simple worked extremely well; after fixing a few bugs during beta-site testing, the January 1980 system was distributed for a full year with no further kernel changes.

Since the initial distribution of the system in January 1980, the use of the system has expanded to over 100 sites. A number of portions of the system have been tuned to increase system efficiency. We feel that the system performs well in our time-sharing environment. The popularity of the system has encouraged its use with the most demanding of application programs and in environments foreign to a time-sharing systems.

We are currently investigating the paging behavior of programs that process very large amounts of data. Large scale mathematical programs and image processing programs tend to have virtual memory behaviors unlike those which have been studied in most of the literature. By exploiting the properties of these programs, it is hoped that the system will be able monitor their behavior and adapt the system's paging policy to run them more efficiently.

Summary and conclusions

A page replacement algorithm that is to function in a machine lacking reference bits must use a minimum of reference information because such information is expensive to gather. The global clock paging algorithm appears to satisfy this condition.

System performance under extreme paging load can be as good using the global clock algorithm as it is using a hybrid paging technique. In practice, the ability of the clock algorithm to vary the memory partitions dynamically increases memory utilization significantly over a scheme which allocates fixed partitions.

The global clock page replacement algorithm is limited in its ability to supply pages on a machine with no reference bits. This is normally not a problem under a time-sharing load, but can be when high data rate programs are run.

Acknowledgments. We are grateful to Domenico Ferrari, Juan Porcar, Jehan-François Pâris, Tom B. London and John F. Reiser for their contribution to the early design effort, and to Charles Roberts, Richard Fateman, Robert Fabry, William Rowan and M. Kirk McKusick for their advice and support.

References

[BABA 80] Ö. Babaoğlu, ''Analysis of a Class of Hybrid Page Replacement Policies,'' _Collection Seminaires INRIA, Modelisation et Evaluation des Systemes Informatique_, 1980, pp. 289-317.

[BABA 81a] Ö. Babaoğlu, ''Virtual Storage Management in the Absence of Reference Bits,'' Ph.D. Thesis, Computer Science Division, University of California, Berkeley, November 1981.

[BABA 81b] Ö. Babaoğlu, ''Memory Management as Inventory Control,'' in preparation, 1981.

[BELA 66] L. A. Belady, ''A Study of Replacement Algorithms for a Virtual Storage Computer,'' _IBM Syst. J._, vol. 5, pp. 78-101, 1966.

[BOBR 72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson, ''TENEX, a Paged Time Sharing System for the PDP-10,'' _Comm. ACM_, vol. 15, March 1972, pp. 135-143.

[CHU 76] W. W. Chu and H. Opderbeck, ''Program Behavior and the Page Fault Frequency Replacement Algorithm,'' _Computer_, vol. 9, November 1976, pp. 29-38.

[COFF 73] E. G. Coffman and P.J. Denning, _Operating Systems Theory_, Prentice-Hall, Enlewood Cliff, New Jersey, 1973.

[CORB 68] F. J. Corbato, ''A Paging Experiment with the Multics System,'' Project MAC Memo MAC-M-384, Mass. Inst. of Tech., July 1968, published in _In Honor of P. M. Morse_, MIT Press 1969, pp. 217-228.

[DEC 80] Digital Equipment Corporation, ''VAX/VMS Internals and Data Structures,'' Preliminary version AA-K785A-TE, November 1980.

[DENN 68a] P. J. Denning, ''The Working Set Model for Program Behavior,'' Comm. ACM, vol. 11, pp. 323-333, May 1968.

[DENN 68b] P. J. Denning, ''Thrashing: It's Causes and Prevention,'' _Proc. Fall Joint Comptr. Conf._, 1968, pp. 915-922.

[DENN 76] P. J. Denning, K. C. Kahn, J. Leroudier, D. Potier and R. Suri, ''Optimal Multiprogramming,'' _Acta Informatica_, vol. 7, 1976, pp. 197-216.

[DENN 77] P. J. Denning and K. Kahn, ''An L=S Criterion for Optimal Multi-Programming,'' _Proc. Int. Symp. on Computer Performance Modeling Measurement and Evaluation_, Cambridge, Mass., August 1977, pp. 219-229.

[EAST 79] M. Easton and P. A. Franaszek, ''Use Bit Scanning in Replacement Decisions,'' _IEEE Trans. Comptrs._, vol. C-28, February 1979, pp. 133-141.

[GRIT 75] D. H. Grit and R. Y. Kain, ''An Analysis of the Use Bit Page Replacement Algorithm,'' _Proc. ACM Ann. Conf._ Minneapolis, Minn., 1975, pp. 187-192.

[JOY 80] W. N. Joy, ''Comments on the Performance of UNIX on the VAX,'' Computer Science Division internal report, University of California, Berkeley, 1980.

[KASH 80] D. Kashtan, ''VMS and UNIX: A Performance Comparison,'' Stanford Research Institute internal memorandum, February 1980.

[KING 71] W. F. King III, ''Analysis of Demand Paging Algorithms,'' _Proc. IFIPS Congress_, Ljubljana, Yugoslavia, 1971, pp. TA-3-155 - TA-3-159.

[LAZO 79] E. Lazowska, ''The Benchmarking, Tuning and Analytic Modeling of VAX/VMS,'' _Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems_, Boulder, Colorado, August 13-15, 1979, pp. 57-64.

[OLIV 74] N. A. Oliver, ''Experimental Data on Page Replacement Algorithm,'' _Proc. NCC_, 1974, pp. 179-184.

[SMIT 80] A. J. Smith, ''Multiprogramming and Memory Contention,'' _Software- Practice and Experience_, vol. 10, July 1980, pp. 531-552.

[SMIT 81] A. J. Smith, ''Internal Scheduling and Memory Contention,'' IEEE Trans. on Software Engineering, vol. SE-7, January 1981, pp. 135-146.