AN ALGORITHM FOR DRUM STORAGE MANAGEMENT IN TIME-SHARING SYSTEMS

Mark L. Greenberg

University of California, Berkeley

An algorithm for efficiently managing the transfer of pages of information between main and secondary memory is developed and analyzed. The alogorithm applies to time-shared computer systems that use rotating magnetic drums as secondary storage devices. The algorithm is designed to provide efficient system performance when referenced pages are predominantly pre-loaded. However, the algorithm also provides optimum results for systems where all pages are loaded on demand. The nature of the improved performance which can be derived from page pre-loading strategy is discussed. Simulation results are presented which plot system performance as a function of main memory size.

Introduction

The efficient transfer of information between main and secondary memory is of major importance in providing a high level of performance in time-shared computer systems. Programs cannot run unless the information they reference is resident in main memory. Yet, due to the limited size of main memory, only a small fraction of the total information stored in the system can be resident in main memory at any particular time. Thus, a heavy load of information transfer into and out of main memory is inevitable.

In this paper an algorithm for efficiently managing information transfer operations in a time-sharing system will be developed and analyzed. Simulation results using this algorithm will be presented. The secondary storage of the system is implemented with high performance, fixed head, rotating magnetic storage devices hereinafter referred to as drums. The algorithm will be developed on the assumption that pre-loading of pages referenced by a running process is the predominant method of page loading. Arguments will be presented to show that page pre-loading can lead to much higher levels of system performance for many hardware configurations. A sufficiently detailed model for such time-sharing systems will now be developed.

Preliminaries

We define throughput of a time-sharing system to be the rate at which user program requests for service are satisfied. In this paper we will consider throughput to be the measure for system performance. The throughput of a time-sharing system is a function of three basic factors: the hardware characteristics, the load of user programs requesting service, and the policies used to determine the resource management operations which convert user requests into actual service.

The hardware model for the class of time-sharing systems considered in this paper consists of main memory, a single processor (CPU), and a single drum with a channel. All main and drum memory is logically divided into fixed size blocks, identical in size to the pages of information manipulated by the operating system. The processor and channel can both access all main memory. There exist M blocks of main memory available for information transfer operations. In a real system there will generally be a number of blocks of memory which are permanently allocated and are, therefore, not part of the M blocks of available main memory. The drum is organized with S sectors around its circumference. One page of information can be transferred during each sector time (the time it takes for a sector to pass under the drum heads). During any sector time, the channel can read from the drum a page of information stored at the current sector position and load it into a block of main memory, unload a block of main memory and write the page of information on the drum at the current sector position, or idle. It is assumed that the drum can switch between read and write operations at consecutive sector times. It is further assumed that simultaneous processor and channnel activity does not significantly degrade the performance of either.

To satisfy a user program request for service, the operating system scheduler must schedule a process at which time the pages required by the process are loaded into main memory. In order to satisfy response time requirements of other programs, the scheduler may preempt a process (and reschedule it later). At this time the pages modified during execution must be written back onto the drum, thus making room in main memory for other pages to be loaded. We define the swapper as that part of the operating system responsible for managing these page transfer operations.

Associated with each process in the system at any given time is a working set. A working set of a process is that set of pages which should be main memory resident for the process to run efficiently. Thus, the pages judged likely to be referenced by the process in the near future should be in the working set. We intentionally avoid discussing in this paper how this judgement is made as the subject has been discussed elsewhere in the literature .

For the purposes of analysis, we can consider a time-sharing system as operating somewhere between two extremes in page loading policy: pure demand paging and pure pre-loading. Many current systems have adopted the pure demand paging policy. Pure pre-loading cannot, in general, be achieved in a real system since working sets can rarely describe the imminent needs of a process with complete accuracy.

For the purposes of this paper, we can think of the load of user program requests for service as being manifested as a stream of processes requesting processor and main memory resources. At any time we are concerned only with those processes which are scheduled and/or have pages of their working sets resident in main memory. Such a process may be in one of four states at any time: the request state, the loading state, the ready state, or the unloading state. A process enters the request state when it is scheduled by the operating system. A process is activated and moved into the loading state during which time the swapper loads the pages of the process's pre-load set. When this is completed, a process enters the ready state. For simplicity we stipulate that only a process in the ready state (i.e. working set totally resident) may run on the CPU. If a running process demands a page, it returns to the loading state until the demanded page is loaded. We refer to the processes in the loading and ready states as the active processes. When a process completes a computation or is preempted by the scheduler the process enters the unloading state while all pages modified during execution are written back onto the drum. We assume that a page may be written on the drum into any free block convenient to the swapper. One sequence of transitions by a process through these states will be refered to as a schedule period of the process.

The main memory allocation strategy adopted in this paper is very simple. Pages which are members of the working sets of loading process should be loaded at the convenience of the swapper into any free block of memory. If a page resident in main memory is not in the working set of a loading or ready process, it may be replaced. It is assumed that no pages of a processes working set are resident in main memory when the process is activated. It is further assumed that there are no shared pages, that is, no page is the member of more than one working set. If either of these assumptions is violated then system performance will improve since less channel time is required to load these working sets.

The Swapper

The swapper proposed in this paper must make two basic decisions:

(1) Process Activation. When and what process to move from the request state into the loading state and thus, into consideration by the swapper. The swapper uses this decision capabilty to limit the number of processes which may simultaneously reside in main memory, thus controlling main memory overcrowding.

(2) Sector Command Selection. Which page to read or write at each sector position of the drum.

There exists one sector queue for each sector of the drum. A sector queue contains a list of all pages which should be read from the corresponding sector on the drum. When a process is activated (the process is moved to the loading state) and entries for all pages in the pre-load set of the process are added to the appropriate sector queues. The demanded page of a running process is also entered on a sector queue. A page entry is removed from a sector queue when the page is loaded. The drum load at any time is defined as the total number of page entries on all the S sector queues. There exists a single page write queue which lists all pages which should be written out. When a process dismisses (completes a computation or is preempted) all pages of its working set which have been modified are added to the page write queue. A page is removed from the page write queue when it is written on the drum.

The swapper must also consider the state of main memory in making its decisions. Each block of main memory is either free or allocated. A block is free if it does not contain a loaded page of a process in the loading state, a page of a process in the ready state or a modified and not yet unloaded page of a process in the unloading state. A block becomes free for each unmodified page in the working set of a process when the process dismisses and for each modified page when the page is unloaded. A block is allocated when a page from the drum is loaded into it.

Sector Command Selection

At a particular sector position the swapper may select to read any page on the current sector queue or write some page on the page write queue. The swapper may select a page read only if there exists a page of free memory to read into. It should make its selection so as to maximize the total drum utilization and minimize the average memory space-time utilized by the processes in the loading and unloading states.

If there is a choice between doing a read or a write, the read should be selected. Let Qw be the number of pages on the page write queue and Qr be the number of resident pages of the process of the page that would be read. If the write is done (and the read not) then the read cannot be done again for a full drum revolution. Thus, the total memory space-time is increased by a factor S*Qr. If the read is done then all the writes are put off only one sector time so the total space-time increases by Qw. Thus, a read should be preferred over a write whenever Qr*S>Qw. This inequality will almost always be satisfied, since S is generally quite large. Thus, the swapper will do a read whenever the current sector queue is not empty and there exists a page of free memory.

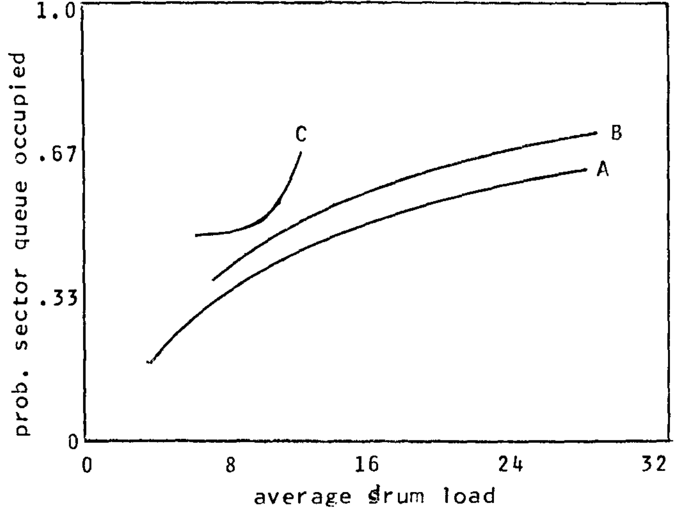
If there is more than one page on the current sector queue then the swapper must decide which page to read. Each page on the queue is a member of the working set of some process. All the processes in loading state are assigned priorities according to some rule. Of the processes with pages on the current sector queue, the swapper selects to do a read of a page of the process with highest priority. If processes are assigned priorities according to the order they enter the loading state, then the sector queues are first come-first served. The main memory required by the loading processes is minimized if the average number of main memory resident pages per loading process is minimized. This suggests the following rule for assigning priorities: priorities are assigned as a function of time according to the number of pages a process has resident in main memory, the process with the most resident pages receiving highest priority. Simulations were made to compare this priority rule against the first come-first served rule and other rules. All the rules yielded approximately the same level of performance with the number of resident pages rule slightly out performing the others. This insensitivity of system performance to the priority rule can be easily understood. The process priorities only affect the swapper function when the current sector queue contains more than a single element and, under the drum loads which produce optimum results, this occurs only a small fraction of the time.

If there does not exist a block of free memory into which to read the selected page, then it may be desirable to release a loaded page of another process in the loading state thus freeing up the block of memory in which the released page resided. The released page must be reloaded later. A page may be released if it is an unmodified resident member of the working set of some process in the' loading state other than the process of the page selected to read. The swapper may release a page only if such a page exists. If more than one such page exists, then the swapper must select which page to release. The same process priorities used to select the page to read are used to select the page to release. The swapper should select to release a page from the process with lowest priority, and the swapper should release the page only if the priority of the process of the page selected to read is higher than the priority of the process of the page selected to release. In other words, a page already resident in main memory should be released and overwritten with another page only if the residency of the new page is considered more important than that of the old page. Introducing this page releasing mechanism into the algorithm allows for greater system performance by permitting a drum load greater than the free memory size without fear of hanging up the system if this should lead to a situation where free memory is exhausted and there are no pages to be written out.

Considerable improvement in the basic algorithm can be achieved if the following drum allocation requirement is enforced. Never allocate any two pages of the same working set at the same sector position on the drum. This implies that the working set size must be less than the number of sectors about the drum, but this will generally be the case with high performance drums. When a working set is so allocated it is said to have no sector conflicts. This allocation requirement is enforced whenever a modified page is written out. A page may not be written out anywhere, but is restricted to those sectors which do not already contain another page of the same working set. Only if the working set size approaches the number of sectors about the drum will there be a significant increase in memory resources required to unload processes as a result of the increased difficulty in finding a sector at which a page of the working set has not already been allocated.

By allocating a working set without sector conflicts it is possible to load the entire working set within one drum revolution time. This optimum reading time can be impeded by an insufficiency of free memory into which to load the working set or by sector interference due to the loading of pages belonging to other processes. Using the no sector conflict allocation strategy, the sector interference problem is reduced. The number of pages on a sector queue, which is proportional to the level of interference, cannot exceed the number of processes in the loading state. With random allocation the number of pages on a sector queue is limited only by the drum load. The effect is to spread the pages on the sector queues out over a greater number of different queues, thus reducing the number of unoccupied queues for a given drum load. System performance for a given average drum load is increased since the effective drum utilization is directly proportional to the probability a current sector queue is occupied (see analysis section). Figure 1 plots the probability that a current sector queue is occupied vs. average drum load (L) for the random and no sector conflict allocation situations. In both simulations the memory size was large enough to avoid page releasing. Notice that for a given average drum load, a greater probability of an occupied current sector queue and consequently a greater throughput is achieved in the case when no sector conflict allocation

Figure 1. Drum Allocation Rules



A - random allocation
B - no sector conflict allocation
C - no sector interference selection
M=32 S=32 w=8 m=4 t=12

is used than in the case when it is not. The curves clearly indicate the advantage of no sector conflict allocation.

When shared pages exist in a time-sharing system, it may not be possible to write out the pages of a working set so as to avoid sector conflicts, since a page which is shared by two active processes will be unloaded when the second process dismisses and will allocate the page without regard to the allocation of the working set of the first process. Of course, this problem does not arise for read-only pages which are the most common kind of shared pages. Writeable shared pages can, therefore, hinder system performance by introducing sector conflicts if they occur in sufficient quantities. It is not believed that this is a serious problem.

It must be further noted that the swapper can do a page write only if there exists an unallocated drum block at the current sector position. Unless the drum is overloaded a free block will almost always exist when it is needed. In all simulation results reported in this paper it is assumed that free blocks exist at all sector positions on the drum.

The complete algorithm is now summarized. The algorithm is invoked once for each sector time.

<u>Sector Command Selection Algorithm</u>

Step 1. (page to read?) Go to step 7 if the current sector queue is empty.

Step 2. (select a read) Of the processes with pages on the current sector queue, select to read the page of the process with the highest priority.

Step 3. (free memory?) Go to step 6 if there is free memory.

Step 4. (select a release) Of the processes in the loading state, other than the one of the selected read, which have at least one unmodified resident page, select a page from the process with the lowest priority. Go to step 7 if no such process exists.

Step 5. (do a release) Release the selected page if the priority of the process of the selected release is lower than the priority of the process of the selected read else go to step 7.

Step 6. (do the read) Start the selected read command and exit.

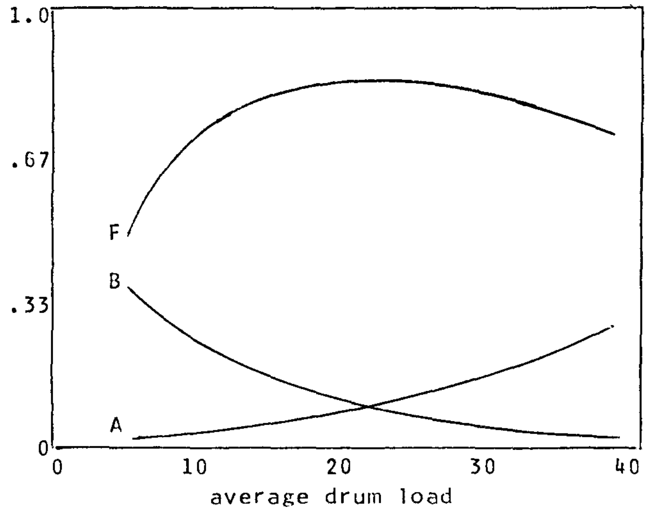
Step 7. (do a write) If a free drum block exists at the current sector, start a write command for the first page on the page write queue which would not cause a sector conflict if such a page exists and exit.

<u>Analysis</u>

It is the purpose of the swapper to maximize system throughput which is directly proportional to the rate at which processes can be serviced by the swapper. If D is the fraction of time the drum is reading or writing pages (i.e. not idle) and U is the average time a process utilizes the drum during a schedule period of a process, then this service rate is just the ratio D/U. Now, let w be the average pre-load set size, m the average number of pages modified during a schedule period of a process and u be the average number of pages released per process schedule period, then, normalizing time units so that a sector time is unity, we have U=w+m+u. D is the probability of reading a page without releasing a page(P_w) plus the probability of reading a page after releasing a page (P_u) plus the probability of writing a modified page (P_m), ($D=P_w+P_m+P_u$). Observing that $P_m=(m/w)P_w$ and $P_u=(u/w)P_w$, we determine that $D/U=P_w/w$. System performance is directly proportional to the probability of doing a useful read (P_w). This probability is in turn equal to the probability of an occupied current sector queue times the probability of the existence of a block of free memory.

The value (w+m)D/U, which we define as the effective drum utilization (F), provides a more convenient measure which is directly proportional to throughput. Maximum system performance is achieved when the effective drum utilization is unity. A little algebra indicates that F is the probability of doing a useful read (P_w) plus that of doing a write (P_m). F falls below unity due to useless reads (reads of pages subsequently released) and idle sector times. System performance is optimized when the summed probability of these latter two events is minimized. Figure 2 plots a characteristic curve for effective drum utilization (F) vs. average drum load (L). System performance is optimized at a particular value for the average drum load. If L is less than the optimum, then many sector queues will be empty and there will be increased drum idle time. If L is greater than the optimum then main memory will contain the pages of more different working sets, thus overcrowding main memory and decreasing the probability of free memory while increasing

Figure 2. Optimum Drum Load



A - probability of page releasing
B - probability of idle sector time
F - effective drum utilization

the probability of an occuppied sector queue, therefore increasing the probability of useless reads due to page releasing. Simulation results for typical systems indicate that optimum system performance is achieved when the average drum load is about equal to half the number of sectors about the drum (L=S/2).

Process Activation

It is through the process activation decision that the swapper is able to regulate the drum load. Initially, the request state will be regarded as a first come-first served queue. The first process in the request state queue is the next process to be activated. This assumption assures a minimum variation in the response times of the processes requesting service. The simplest activation algorithm is to consider the next process whenever adding its pages on the sector queues would not cause the drum load to exceed some value specified as a parameter to the algorithm (L'). In this case the average drum load would be given approximately by L=L'-w/2. The drum load would vary with the variation in pre-load set sizes.

This algorithm can be improved upon if the amount of free memory is taken into account in making the process activation decision. When free memory is larger, a higher drum load can be tolerated, thus suggesting the following activation algorithm: activate the next process if the drum load plus the size of the pre-load set of the next process minus the amount of free memory is less than the parameter L'. The approximate average drum load would then be L=L'-w/2+Mf where Mf is the average free memory size. This modification tends to make the probability of activating a process higher if there is more free memory. This, in effect, introduces feedback into the queue of free memory blocks by increasing the arrival rate into the queue when the free memory size is small. A smaller free memory size will cause a smaller drum load, which will reduce the probability of doing a read, which will increase the probability of doing a write, which will increase the arrival rate into the free memory queue. Thus, on the average less free memory is required, therefore, more memory is available for loading processes which allows a higher average drum load and consequently a greater level of system performance.

If processes are taken off the request state queue in some order other than first come-first served, then the activation algorithm can be improved even further, but not without cost. The following rule when added to the previously specified algorithm can completely eliminate sector interference. Select as the next process to activate the first process on the request state queue whose pre-load set if added to the sector queues will not add a page entry to any sector queue which is already occupied. A process may be activated only if such a process exists in the request state. Using this sector interference elimination rule, a current sector queue can have just one or zero pages in it. The effect is to maximally spread the pages over the sector queues. System performance is increased by increasing the probability of an occupied current sector queue for a given average drum load. Typical simulation results showing the improvement due to this process selection rule are plotted in figure 1.

This process selection rule has three disadvantages. First, it requires additional computational overhead to determine if each process in the request state causes any sector interference. Second, since the request state may be serviced in any order, the variation in response times will increase considerably. In fact, it cannot be guaranteed that a process in the request state will ever be serviced. Only systems with sufficiently long response time requirements can afford to use this rule. Third, a system performs better under this algorithm only if the request state is sufficiently large. There is a greater probability of finding a process which causes no interference if the request state is larger. Thus, the average drum load is a monotonically increasing function of the average request state size. If a time-sharing system is not sufficiently loaded (this may be because excessive loading would cause unacceptable response time) to produce a sufficiently large request state, then the average drum load might be so low that better performance could be achieved with first come-first served selection. For example, in the situation simulated in figure 1, curve C it was found that this selection rule increases performance over first come-first served selection only if the average request state size is greater than seven.

Using this sector interference elimination selection rule, system performance can be improved still further by restricting the allocation of pages of a working set to some contiguous subsection of the sectors of the drum. In doing this it becomes easier to find a process which does not cause any sector interference for the same request state size since the sector positions of the pages of two different working sets are more likely to be disjoint. Alternately, it can be stated that this allocation restriction improves performance by reducing the time it takes to load a working set. This allocation restriction helps only if the average pre-load set size is much less than the number of sectors about the drum(w<<S) and writeable shared pages are not so prevalent as to nullify the effect of this allocation. If w is too large compared to S, then the increased difficulty in writing pages due to the allocation restriction will cause the average amount of memory required by processes in the unloading state to be increased by more than the loading state memory requirement is reduced. As an example, simulation results show that if the average pre-load set size is about one-fourth the number of sectors about the drum (w=S/4), then a significant (5-10%) improvement in system performance occurs when the pages of a working set are restricted to one-half to three-quarters of the total drum sectors.

Simulation

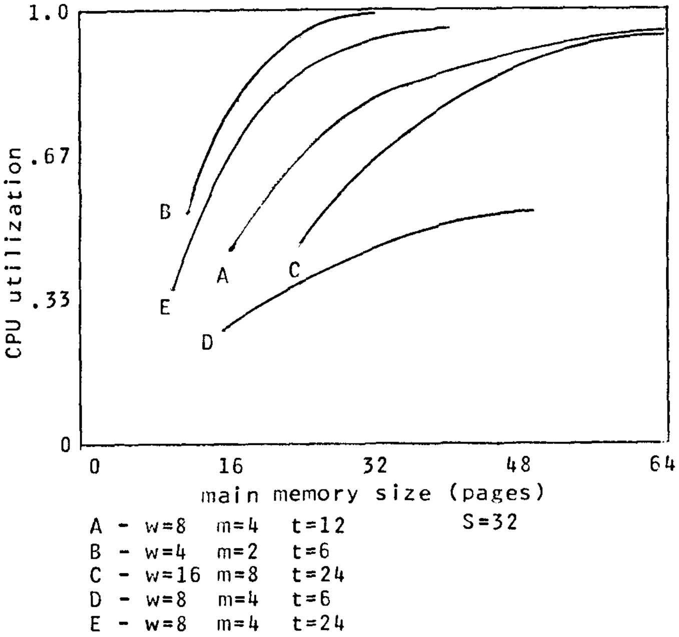
A simulator was developed to determine system performance and other measures as a function of the system parameters under certain assumptions now stated. The simulator operates in descrete time units, the atomic unit being a sector time. In all simulation results presented in this paper, the compute times of processes during schedule periods are Poisson distributed with mean t, and the pre-load set sizes are uniformly distributed between w/2 and 3w/2. The number of pages of a process modified during execution is always directly proportional to the working set size of the process in the ratio m/w or m/d. For pure demand paging conditions, the number of pages demanded by a process is proportional to its compute time in the ratio w/t and all pages are demanded at compute time zero of the schedule period. All simulation results reported in this paper are for a system with one CPU and one drum with 32 sectors. The simulator was implemented in QSPL^4, a system programming language for the Berkeley SDS-940 time-sharing system^5.

Figure 3 plots simulation results which compare CPU utilization against main memory size for systems with several different program loads under pure pre-loading strategy. Curves A, B, and C correspond to systems with balanced loads, curve D to a channel limited system, and curve E to a CPU limited system. CPU utilization is directly proportional to effective drum utilization in the ratio t/(w+m). Remember t is measured in units of sector times. In all these simulations the system is always fully loaded i.e. the request state is never empty.

Operating System Overhead

The swapper requires computational and memory resources in order to function. It requires memory to hold the sector queues, the page write queue, and the code of the swapper program. This amount of memory should be very small compared to that needed to swap working sets. In addition, there must be memory for the operating system data structures which locate and identify pages on the drum and in main memory. This memory requirement can be quite large in some operating system designs.

Figure 3. Pure Pre-loading Performance



The computational requirements of the algorithm can be quite high. The swapper procedure must be called once every sector time of the drum. If the page size is small, say 512 words, then the procedure might be called as frequently as once every 250 micro-seconds. The computational overhead is inversely proportional to the page size. In systems with one or more large drums it is probably necessary to dedicate an entire processor to the swapper function since the swapper decisions require non-trivial computations. Because of the swappers specialized function, it might be most efficiently implemented on a micro-processor.

Demand Paging

The swapper using the previously described algorithm can satisfy demand paging requests of running processes in a very natural way. When a process demands a page, the process is immediately moved to the loading state and the demanded page entered on the appropriate sector queue. Thus, the swapper sees a process whose working set is loaded but for a single page. Even in the extreme case where all pages are loaded by demand paging, the algorithm works correctly and efficiently. In fact, using the number of main memory resident pages to determine process priorities provides a better performing swapper than the 'paging drum' described in the literature^{1,3} (which uses first come-first served priorities) by minimizing the memory required by processes in the loading state.

Under pure demand paging strategy, a different rule should be used to determine when to activate a process. Unlike the pure pre-loading case (where it is assumed that all pages to be referenced are pre-loaded), it is not known at activation time what the eventual main memory demand of the activated process will be. Therefore, under pure demand paging a new process should be activated only if the sum of the expected memory demands of the active processes (i.e. in the loading and ready states) does not exceed some parameter whose value is proportional to the main memory size. The constant of proportionality can be tuned to achieve an optimum compromise between drum utilization and page releasing resulting from an overcrowded memory. The working set size for a process can serve as an estimate for the expected memory demand. Lacking this information, the estimate for a process's memory demand can just be the average number of pages demanded by processes in the system. This amounts to maintaining a constant number of active processes at any time i.e. a process is activated whenever another process dismisses. This is the activation rule used in the reported pure demand paging simulations.

Pre-loading vs. Demand Paging

In this section we will present arguments based on simulation results and approximate analysis which will demonstrate the improved performance which can be achieved by using pure pre-loading strategy instead of pure demand paging strategy. Let w be the average pre-load set size under pure pre-loading and let d be the average number of pages demanded by a process during a schedule period under pure demand paging. The extreme of pure pre-loading cannot be achieved in a real system, but as this limit is approached the average pre-load set size will tend to contain more pages which are not subsequently referenced when the program runs during the current schedule period. Clearly, the average pre-load set size under pure pre-loading is greater than the average number of pages demanded per schedule period under pure demand paging (w>d), however, a precise relationship between these two quantities can be determined only through empirical evidence obtained from real systems.

At any time each block of main memory is either free or allocated to a process in the loading, ready, or unloading state. Therefore, we can define four quantities which represent the number of free main memory blocks or the number of main memory blocks allocated to all the processes in these three states at any time averaged over all time. We call these quantities the free, loading state, ready state, and unloading state memory requirements and denote them by M_f , M_{ls}, M_{rs}, M_{us}, respectively.

A numerical example based on simulation results will be instructive. Assume a loaded (request state never empty) and balanced (w+m=t) system and a drum with thirty-two sectors. Further assume a system load with average working set size of eight pages (w=d=8), half of the pages modified during the schedule period (m=4) and mean compute time per schedule period of twelve sector times (t=12). Under these conditions the following simulation results are obtained for the pure pre-loading case: $M_{ls}=16$, $M_{rs}=8$, $M_{us}=3$ and $M_f=3$ for a total memory requirement of 32 blocks. These results are obtained with an average drum load of 16 pages and a resulting optimum effective drum utilization of about 80%. Under pure demand paging the following simulation results are obtained: $M_{ls}=42$, $M_{rs}=8$, $M_{us}=3$, $M_f=15$ for a total memory requirement of 68 blocks with approximately the same average drum load and optimum effective drum utilization as before. Thus, for this example about twice as much memory is required under pure demand paging to achieve a given level of system performance as under pure pre-loading.

We will now provide arguments which explain this difference in memory requirements. A little reflection should convince the reader that the ready state and unloading state memory requirements will be approximately the same for a given level of system performance regardless of the loading strategy used. The average free memory size will, in general, be greater for the demand paging case as a result of the greater uncertainty of the future memory demand of an active process.

The loading state memory requirement shows the greatest dependence on the page loading strategy employed. Under pure pre-loading at any time each of the pages of a process in the loading state is either resident in main memory or on a sector queue. If there is no sector interference between pages of different processes, then all pages in the sector queues will be read at the first opportunity and therefore, assuming uniform distribution of pages around the drum, we conclude that a loading process has an average of w/2 pages resident in main memory ($W_{ls}=w/2$) and w/2 pages on the sector queues. The average number of processes in the loading state is just the average drum load divided by the average number of sector queue entries per process ($Q_{ls}=2L/w$). The loading state memory requirement is the product of the average number of resident pages per loading state process times the average number of such processes ($M_{ls}=W_{ls}*Q_{ls}$). Therefore, we conclude that the approximate loading state memory requirement under pure pre-loading is just the average drum load ($M_{ls}=L$).

Under pure demand paging there is, in general, one page on a sector queue for each process in the loading state. Therefore, the mean loading state size is approximately the average drum load ($Q_{ls}=L$). The average number of resident pages per process in the loading state is approximately half the number of pages demanded per schedule period assuming the pages are loaded uniformly with time ($W_{ls}=d/2$). Multiplying these two factors we get $M_{ls}=Ld/2$ as the approximate loading state memory requirement under pure demand paging. The loading state memory requirement of 42 pages determined from the previous simulation result is considerably less than the 64 pages expected from this approximate analysis. This reduction can be largely attributed to the advantageous use of the priority rule based on the number of resident pages of a process.

A system is channel limited if the average drum channel time utilized per schedule period is greater than the average compute time per schedule period (w+m>t). Assuming that the system is channel limited (which is the only interesting case), we argue that the throughput of a system is the same for a given average drum load if the system is operating under pure pre-loading or under pure demand paging. Since less pages must be loaded per schedule period under pure demand paging (d<w), a greater throughput is achieved for a given effective drum utilization. However, a greater effective drum utilization is achieved for a given average drum load under pure pre-loading due to the no sector conflict drum allocation strategy. In figure 1 the difference is clearly indicated where curve A corresponds to the demand paging case and curve B corresponds to a pre-loading case. These two effects approximately cancel yielding about the same throughput for a given drum load under either loading strategy. Therefore, using the results of the previous two paragraphs, we conclude that the loading state memory requirement under pure demand paging is greater than that under pure pre-loading by a factor of about d/2.

It should be noted that the reduced memory requirement due to pre-loading can be obtained only in systems with drums with a large number of sectors. The average drum load which produces the optimum performance for a particular program load will be proportional to the number of drum sectors. The reduced drum load for smaller drums will reduce the loading state memory requirement proportionally under both strategies and consequently the advantage of pre-loading is diminished.

Summary

Three things were accomplished in this paper: 1) we have developed and analyzed an algorithm for efficiently managing the transfer of pages between main memory and a large secondary drum storage device in systems which use either demand paging or pre-loading page loading strategies, 2) simulation results were presented which plotted system performance against main memory size for a system using pure pre-loading strategy under various program loads and 3) an argument based on simulation results and approximate average value analysis was presented which concluded that the use of pre-loading strategy with the given algorithm instead of pure demand paging can considerably reduce the main memory required to achieve a particular level of system performance if the drum is large enough to store considerably more than a typical working set of information in one drum revolution.

Acknowledgement

The algorithm described in this paper was initially developed by the author and Robert R. Van Tuyl during the summer of 1968[7]. The author wishes to thank Charles A. Grant for his helpful advice concerning the preparation of this paper.

References

1. Denning, P.J., Effects of Scheduling on File Memory Operations, SJCC, 1967, pp. 9-21.

2. Denning, P.J., The Working Set Model for Program Behaviour, Comm. ACM, 11, May 1968, pp. 323-333.

3. Denning, P.J., Virtual Memory, Comp. Surv., 2, Sept. 1970, pp.153-189.

4. Deutsch, L.P., Lampson, B.W., QSPL Reference Manual, Document R-28, ARPA Contract SD-185, University of California, Berkeley.

5. Deutsch, L.P., Durham, L., Lampson, B.W., Reference Manual Time-Sharing System, Document R-21, ARPA Contract SD-185, University of California, Berkeley.

6. Greenberg, M.L., Secondary Storage Management in Time-Sharing Systems, Ph.D Thesis, Dept. of EECS, University of California, Berkeley.

7. Van Tuyl, R.R., An Algorithm for Swapping Data from Drum to Core, Document P-16, ARPA Contract SD-185, University of California, Berkeley.