To: Distribution

From: Steve Webber

Subject: New Page Control Design Proposals

Date: July 8, 1974

Introduction

This MTB discusses some of the proposed changes to the page control organization of Multics. The changes are extensive and constitute a considerable deviation from the current structure and algorithms. The justification for the changes comes from several sources but primarily our own metering and analysis of the current system with its current load characteristics. Some of the basic problems the new scheme hopes to solve are:

1) uniformity of throughput so that system efficiency does not degrade as load increases

2) more equitable core accounting - the current memory units scheme just doesn't work well.

3) potentially more efficient algorithms which partially distribute the global paging lock and therefore make multiple CPU configurations more efficient.

The new design for page control differs from the current design primarily in the core management algorithms. The "core control" functions are to be split apart from the page fault handling functions thereby giving us more freedom in the choice of core removal algorithms as well as allowing us to partition the code into separate "tasks".

Basic Assumptions

One of the main reasons, today, for our excessive overhead caused by the paging mechanism is that user processes have strikingly large working sets. This is made even worse by the fact that most processes change the contents of their working set quite rapidly. A result of these characteristics is that a great many pages are brought into core, referenced for a short period of time and then not referenced for a long time - long enough to have the page or pages removed from core. Whether this behavior is inherent in a system like Multics, or in a user

---

community such as MIT or Phoenix, or whether it will continue to behave in this manner are interesting questions. We, however, must look at the more immediate problem of trying to find a paging system that works well in this environment as well as trying to understand the reasons for the behavior. (Indeed, another task being undertaken is just this study of the causes of this behavior with the intent of possibly changing the basic designs and constructs.) It is much easier to provide page removal algorithms for slowly changing working sets (swapping is an extreme). The new paging system we hope to develop should work well with either kind of system behavior.

With the rapidly changing working sets and pages referenced for only a short period of time comes the interesting result that a great deal of main memory is apparently not being used. This has been observed and verified by several different schemes of sampling and metering the MIT system. In fact, usually there are about 30-50 percent of the pages in use which have not been referenced in the last "lap" - where a lap takes from one-half to one second. It is these pages which are candidates for removal. It should be noted that there are <u>many</u> pages which are good candidates for removal. It is a buyer's marker for core blocks (today!).

Another observation to be made is that since there are a great many pages which are referenced for only a short time after they are brought into core it might be worthwhile to sample all pages a given period of time after they were brought in to see if they are still being used. This very experiment was modeled (with a page control/scheduler modeling program) and, indeed, better paging behavior - i.e. fewer page faults - resulted when this was done.

If one analyzes the workings of the current algorithm it is noted that the "lap time" is used as the sampling period and that the lap time changes with the amount of core configured and the removal algorithm used. However, program behavior and reference patterns in particular are independent of configuration (for the most part) and it is therefore unlikely that this algorithm which tries to control one entity with another fairly independent one is optimal.

The experiment performed to verify the above showed that rather than 500 to 1000 millisecond between sampling, 50 millisecond was better. This says that many pages are referenced for up to 50 milliseconds and then not referenced again. The new proposal uses this finding and sets a sampling rate which is independent of the configuration and possibly even load.

One last assumption should be noted. This is that for several reasons it will be beneficial to develop a scheme which adapts well to a disk only system.

Goals

With the above assumptions and with our stated broad goals we are now in a position to propose the general new organization. First, however, more detailed goals will be listed so that it will become clearer why some design decisions have been made.

1) we would like to split up the page control lock mechanism so that independent functions of page control can go on simultaneously,

2) we would like to be able to get some useful work done by the "idle" processes, if possible,

3) we would like to update, in some way, our disk management routines to take better advantage of the hardware. This includes using rotation positional sensing, seek minimization, and the like,

4) we would like to be able to continually monitor the system so that it can automatically adjust to changes in load to ensure throughput is not degraded,

5) we would like to set up communication between the scheduler and page control so that each can work with a broader base of knowledge.

Core Control and Page Stealing

A major change in the new algorithms is the manner in which core control interfaces with page control. Currently, the two are tightly bound together both in design and in data structures. The new scheme will separate the two nearly completely. Page control will contain code to handle page faults and provide paging interfaces for segment control. Core control will manage the core map, decide which pages should be removed from core, etc. The actions of page control will be under control of a global lock, the page table lock. The actions of core control will be under control of a different lock, the core map lock. The names of the locks intentionally include the names of the data structures they are intended to control.

The core algorithm currently is invoked wherever a block of core is needed, either for a faulted page or for a read/write sequence (RWS) used to move a page from the paging device. The new scheme would replace this mechanism with one that continually attempts to keep a pool of free core blocks for use when needed. The obvious disadvantage here is that core, a valuable resource, is apparently being wasted while it sits idle in the free list. However, there are two reasons why this is not as bad as it might seem. First, as noted earlier, under current reference behavior there is a lot of core around not being used which might as well be threaded into a free list. Second, the new algorithm gives us more freedom and flexibility for trying other changes and extensions.

One of the more important changes we would like to try is in the actual management of the core map. Since we are assuming the core management functions are not part of the page fault handling code, the actual work done can be performed completely asynchronously - in another process. This is in fact what is being proposed, i.e. that the core replenishment task be run by whoever notices the need and whenever it is noticed. As a special case, this includes the idle processes, which, on a disk only system may run a considerable percentage of the time.

The actual algorithm of the core replenishment task will be a form of <u>page stealing</u>. Page stealing in this context is nothing more than "continually" searching core for blocks which can be freed. The prime feature is that the rate of stealing can be controlled and the actual implementation will consider the "owner" of the page and weigh the value of removing the page with respect to the paging behavior the owner is exhibiting.

Since the rate of stealing (number of core blocks freed per unit time) is not necessarily the same rate at which page faults eat up core, any imbalance between the two mechanisms will tend to increase or decrease the amount of free core at any instant in time. This quantity, the amount of free core, will be used as the prime factor for controlling eligibility. (In the past, this decision has been made based on the working set estimates of processes calculated when the processes last ran.) By controlling eligibility by the dynamic core requirements of the processes running the instant eligibility is to be awarded, we have a much better chance of success in preventing thrashing. The new eligibility decisions will still take working sets into account but they will use current core demands as a better base from which to make the decision.

The page stealing algorithm will be run by the system at three distinct logical points during normal operation. These are 1) at page fault time, if necessary, 2) when the idle process runs and 3) (most likely) when the process dispatcher runs. The actual program(s) will steal as many pages as seem appropriate as described in the sketch of the general algorithm later on.

<u>Process Page Pools</u>

It has been noted many times that a feature that protected one process against the paging behavior (usually thrashing) of another process would be desirable. This is true especially if we want to be able to support very cheap, limited subsystems such as BASIC or text editing. In order to do this some mechanism for determining which process "owns" a page must be established. Various techniques have been proposed and some attempted. The one proposed here is slightly expensive, although the expense can be administratively controlled. Basically each block of core will be tagged with the process that owns the page residing in the core. For non-shared segments (per-process segments or segments that only one process has referenced) it is safe to set up the owner of a page as the process that faults on the page and brings it into core. About 30% to 50% of all page faults are on such pages. For (potentially) shareable pages, this mechanism doesn't work. The process that faults on a page may easily not be the heaviest user of the page. For such pages the owner is set up initially as the process that faulted on the page and then if the page remains in core for a considerable period of time (a second or two) the owner is changed appropriately. This is done by placing a special fault in the PTW for the page, the handler of which does nothing more than remove the fault, update the owner of the page and update the page pool sizes of the old and new owners. The frequency with which the fault is set (the handler takes from 50 to 100 microseconds) can be set administratively to control overhead at the expense of resolution. With 5 second resolution (which is considered more than adequate and equitable) the overhead is so small it could not be measured (down in the noise).

The owner of a page will therefore be a particular process on the system (or the page will be "free"). This association will be made by placing a unique process tag in each core map entry. (The tag will actually be a pointer to the APT entry for the process.) This ownership quality can be directly used as a means of core (or main memory) accounting. When a page is brought into core a clock reading will be saved in the core map entry associated with the page. When the page is thrown out of core another clock reading will be made and the core residency value will be updated into the APT entry for the process owning the page. The reasons that schemes such as this have not been used (on Multics) in the past is due to 1) the problem of pages remaining in core after they are not needed (usually in an idle system) and 2) the problem of having one process fault on a highly used page and therefore having to pay for it as long as other processes keep the page in core. Page stealing as described below, with the aid of the special fault mentioned above, solves both of these problems.

The fact that each block of core in the system is assigned to a process effectively partitions all of core into distinct "pools of pages". The size of these page pools can be monitored and controlled by the system. A process can be guaranteed a certain minimum number of pages in core and restricted to less than some maximum. In fact, the controlling of the sizes of these page pools will be one of the critical tasks of the scheduler. It is this control that will prevent a runaway process from forcing the pages of another "innocent" process from core. It will be this same control that will allow a large working set process to establish a large page pool and keep it for a long enough period of time to warrant the overhead of running the process at all. It will be up to the scheduler to determine page pool limits within which a process should be constrained while it runs. It will also be up to the scheduler to determine when and how these limits are changed. The algorithms to be used here are sketched in the following section.

The Scheduler/Page Control Interface

It has frequently been claimed that the traffic controller and page control should communicate more. This claim is hereby made again. The prime reason for the claim is that the decision to run a process cannot be merely an administrative priority decision if the system is to perform efficiently. There is a considerable overhead in getting a process going after being blocked (or whatever) in a virtual memory system - especially one like Multics with its large working sets. This overhead must be considered by the scheduler both in the order in which to run processes as well as the length of time a process should be run (i.e. remain eligible and competing actively for core). In the past the scheduler has for the most part ignored this overhead and based all of the decisions on how long the process has run since it "interacted". Although this variable should probably be integrated into the scheduler decisions it should probably not be weighed anuwhere nearly as much if efficiency is to be maintained. Instead, the following items are claimed to be at least as important:

1) the working set (as estimated by page control),

2) the recent paging rate of the process (as measured by page control) and

3) the recent thrashing rate of the process (as measured by page control). The thrashing rate is the ratio of page faults taken in a quantum on pages that were already faulted on in the quantum to the total number of page faults taken in the quantum.

All of these are easy to come by given that we continue our post purging activities. Note the introduction here of a measure of thrashing as a critical quantity here in the scheduling mechanism. This is because thrashing gives us a measure of whether a process really does not fit within the core limits assigned to it as opposed to a process that won't fit in any core no matter how much is assigned. As an example, the backup process takes an extremely large number of page faults as it dumps segments. But all of the faults are on different pages (hence, no thrashing) and the page fault rate would not be decreased no matter how much core was assigned to the process. In fact, the optimal amount of core would be just enough to fit the code and working data of backup plus a few buffer pages to hold data until it could be written onto tape. The large page fault rate of backup could not be helped by more core. On the other hand, a large PL/I compilation may take many page faults on the code of the compiler and the temporary tree structure during a compilation. Here, thrashing would be especially evident if only a small amount of core were allotted to the process. Therefore, the thrashing, it is claimed, is the indicator that should be used when determining when to grow and shrink the core limits of a process. By constraining backup to a page pool of the appropriate size we can aid the core removal algorithm by forcing it to remove one of backup's buffer pages which is no longer needed rather than a potentially usable page of another process.

The page pool limits will, of course, also have administratively controlled constraints which may vary from zero to infinity. Such constraints can be used to override the page control inputs both to force certain processes to have better response (supposedly at the expense of system efficiency) as well as to meter and tune the system and check out modifications to the algorithms.

A second major interface between the scheduler and page control is in the area of process loading and unloading. Currently the loading of a process (i.e. paging in the PDS and DSEG of the process so that it can run and page anything else it needs itself) is triggered by the scheduler when it decides the process should be allowed to run. Similarly, the unloading (releasing or unwiring of the PDS and DSEG) is done when the scheduler or the process itself has decided that the process will not be run again for awhile. In the current system the unloading of a process is accompanied by the "post purging" of the process. This includes looking at all of the pages the process faulted on and brought in during its last quantum (eligibility period). Certain functions are performed depending on what types of pages were faulted on, how long they were used, whether they are still in core, etc. The functions are specified (dynamically if desired) by a set of boolean equations coded into the post purge program. They include:

1) writing out a modified page before it otherwise would be written out,

2) marking the page as not having been used for a long time by turning off the "used" bit in its PTW and/or rethreading the page's core map entry to the "least-recently-used" end of the core map,

3) counting the page in the working set,

4) measuring the thrashing of the process by noting which pages were faulted on more than once in the quantum, and

5) measuring the total page faults for the quantum - and hence the paging rate.

Process Swapping

It is proposed that the post purge function be extended so that all per-process pages in core at process unload time be written out onto a contiguous region of disk (not bulk store). Obviously, these pages would be swapped back into (discontiguous) core at process load time, i.e. when the process is again awarded eligibility. This is analogous to the "pre-paging" technique used on the 645 and made feasible by the high transfer rate and latency optimization that could be pulled off with the DRUM. Both of these features exist in a limited way, with the DSU-191 disks. The transfer time for 1024 words of data is 6.7 milliseconds (the DRUM was 2.1 milliseconds) and with rotational positional sensing the latency can be minimized. The seek time for the disk can be made minimal by allocating all "swap images" in adjacent cylinders on a "scratch" pack. It has been estimated that 200 to 300 users could be swapped in and out with 30 seconds delay per process between swappings if each user had a swap image of between 10 and 20 pages (a reasonable number for the per-process pages of a process not doing something like a PL/I compilation). This estimate would have to be modified downward as a function of the number of "large" processes competing for resources.

A partition of disk will be allocated at bottload time as the SWAP partition and will be divided into swap "images" of a given maximum size. Each APT entry will be assigned one such image (at bootload time) for the life of the bootload.

The dispatcher would decide when a process is to be swapped in (a short time before it is run, supposedly). It would call upon page control to get enough free core blocks and initiate the appropriate "scatter" read into the acquired core. The page swapping program would upon its completion, "connect" the core blocks to the appropriate pages.

When a process is unloaded the scheduler will again call upon page control first to post purge the process (collect statistics, etc.) and then to swap the process out. The swap out mechanism will consist of little more than issuing the appropriate "scatter" write request, saving any necessary information and freeing up the core when the disk I/O is complete.

The concept of process loading will be replaced by the swap in function. The concept of process unloading will analogously be replaced by the swap out function.

Pages that were swapped in may be paged out (to the paging device supposedly) during a quantum. However, any pages that are to be part of a new swap image that are on the paging device will be deleted from the paging device at swap out time. The only (most) valid copy will exist in the swap image. (Only pages in core at swap out time will be assigned to the swap image.) This freeing of paging device records will considerably ease the traffic flow to and from the paging device.

The benefits of this sort of process swapping are fairly clear. The disks are used much more efficiently for the class of pages which can be swapped. The page faults that are avoided by the swap in presumably cost much more than the swap in code. (Much of the cost of handling the fault is verifying that the fault still exists, etc.) By swapping stacks, linkage, KST's, etc. to the disk the pages need not reside on Bulk Store. It has been noted that of the 2000 pages of Bulk Store at MIT about 800 would be freed up if swapping were being done. A system without a Bulk Store would certainly be more efficient if swapping were being done. The swapping mechanism works especially well for the small, tightly coded subsystems that we would like to optimize. The problems with swapping in a process are two fold. First, greater pressure is placed on the core freeing mechanism so that the entire swap image may be brought in at once. This is supposedly not a problem if page stealing is working. Indeed, whether or not to award eligibility and hence swap in a process will be based on whether the free core is available.

The second major problem is that a process may take longer to set up its initial working set by swapping it in from disk rather than paging it in from Bulk Store. Although it is true that a process won't be running (in real time) as soon after it is decided to run the process, the system efficiency will be higher because less CPU time will have been spent to get the same work done. An obvious design is to "preload" a process by initiating the swap in before the process is to be run. Whether or not preloading will be attempted has not been decided (the demand for core is made earlier which may interact with the running processes). Usually a swap in would be scheduled a short time after a swap out so the core freed could be used.

A third difficulty that arises with swapping is the high use that will be made of the disk used to swap with. A single channel will be saturated with 300 users and queuing effects come in to play long before this. Clever schemes may need to be developed to ease the burden on particular disks or disk channels. It very well may be cost effective to purchase another disk subsystem just for swapping.

Note that the swap image on disk will probably contain different pages each time it is written. It is the ability to "choose" the disk address we write a page to that enables us to use the disk in this manner. Several new data structures, some wired down, must be added to the system to enable page control to determine which pages were actually written where into the swap image. It is the task of page control to determine the location of most up to date copy of a page. It may be 1) in core, 2) on the paging device, 3) on the swapping device or 4) on normal disk.

The Page Stealing Algorithm

Page stealing will be done by one and only one program set - core control. Core control is called occasionally to replenish core and as required to provide free core and accept other core as being free. The basic algorithm to be used will be a least-recently-used algorithm modified as noted below. There are several parameters to the removal algorithm which are tunable by the system administrators. Some variables of the removal algorithm are changed by the algorithm itself in an attempt to adapt to changing user load and behavior.

Before the actual algorithm is described the structure of the core map will be briefly described.

The Core Map

The core map consists of a header containing global control information and list pointers followed by an array of core map entries (CME's) indexed by the absolute address of the core associated with the entry (divided by the page size). The entries themselves may be threaded into several lists independent of absolute address.

Associated with each CME are 1) a pointer to the PTW for the page residing in the core block, 2) the device address of the page, 3) the time the entry was last looked at by core control, 4) a pointer to the APT entry of the "owner" of the page, and 5) various control bits and thread pointers.

There are three threaded lists of CME's managed by core control: the free list (FL), the recently faulted list (RFL), and the extended residency list (ERL). The free list is linearly threaded and managed with a LIFO strategy. The RFL contains all CME's for blocks of core recently awarded to a process because of a page fault. The ERL contains all other nonspecial CME's.

In addition to the CME's threaded into the above lists there are other CME's which are threaded into no list. These are:

1) CME's for perm-wired core (core that is not in the paging pool),

2) CME's for blocks of core being used for read/write sequences,

3) CME's for blocks of core that contain temp-wired pages,

4) CME's for blocks of core for which read I/O is going on, and

5) CME's for currently unconfigured core.

The header of the core map will contain the obvious pointers to the lists as well as useful counters such as the number of CME's in each list or state. The header will also contain metering data and control variables used by the removal algorithm.

The actual removal algorithm works as follows:

1) Check the RFL and move any CME's that have been in the list for over alpha seconds to the tail of the ERL turning OFF the page-has-been-used (PHU) bit of the PTW associated with the block of core.

2) Check the head of the ERL and move any CME's that have been used in the last beta seconds to the tail of the ERL turning OFF the PHU bits.

3) Free any unmodified block of core that has not been used in the last beta seconds.

4) Initiate a write request for any page that has been modified at some time but has not been used in the last beta seconds and which has not been written out since it was last modified.

When a page fault occurs, a block of core is taken from the free list and placed at the tail of the RFL. The time of the fault is stored in the CME at this time. The RFL is a linearly threaded list strictly ordered by time of entry in the list. The core removal algorithm searches this list whenever it is invoked and moves as many entries from the head of the list as is appropriate to the tail of the ERL. No entry should remain in the RFL for more than alpha seconds (within the resolution of time between calls to core control).

Similarly, core control looks at the head of the ERL, which is also linearly threaded and strictly ordered by time of last "move" in the list and takes some appropriate action on all CME's that have not been looked at for beta seconds. The important feature is that the rate at which CME's are sampled is a function of alpha and beta and not the number of entries in the core map (i.e. the current lap time). As long as core control is invoked frequently enough this sampling rate will be as constant as alpha and beta (which may be varied).

The core accounting will be done at two places during execution of the removal algorithm. First, when a CME is moved from the

RFL to the ERL and second whenever a CME is moved from the head to the tail (for another cycle) of the ERL. When the page is faulted on, the APT entry pointer of the faulting process is placed in the CME and used to determine the account to which the core residency should be charged. This APT entry pointer specifies who is the "owner" of the page. The removal algorithm will, however, change the owner (by changing the saved APT entry pointer) if a page stays in core for an "extended" period of time (maybe five seconds) and the page cannot be identified as belonging to a single process. In this case, the special fault is set causing the owner to be recalculated when and if the page is ever referenced again. Pages which are not potentially shareable pages are 1) per process pages and 2) pages of segments which only one process is using.

It is clear now how the two problems of core accounting mentioned earlier are solved. First, because a page will be freed soon after it is no longer used the "idle system" problem goes away. Second, by changing the owner of shareable pages that remain in core for an extended period of time after they are faulted on, a user will not have to pay for a page which he brings in but which other processes use after he is through.

An important refinement of the removal algorithm comes into play when the page fault rate is higher than the rate at which core control can free pages on its own. When this occurs, (the free list is empty) the page fault handler calls upon core control (before locking the page table lock) to free up a block of core. This call however, is slightly different in two respects from the standard call upon core control to do what it can. First, core control must find a free core block even if it means looking at CME's which have not been in the ERL for beta seconds. Second, core control, when invoked at page fault time, knows on whose behalf the block of core is to be claimed. In particular, core control can give the process a core block which it already owned - if the process was at or above its page pool size already - or core control could give the process a block which it did not own thereby allowing the process to increase his page pool size. Note that this additional information can be used in exactly the case where it is most needed, i.e. when the system starts to page too heavily.

The values for alpha and beta that are being considered are about 50 and 200 milliseconds respectively. These numbers will of course, have to be optimized experimentally but it should be noted that they were chosen so that core would be sampled for use at least as frequently as today (at MIT) and hence core control should be able to stay ahead of the paging rate.

Further Notes

Two final notes should be mentioned. First, the initial implementation attempted (if and when) will not use more than one lock. The current global page table lock will be used for core control and page control. This means that one of the important design goals will not initially be realized, but it also means that a working version will be available much earlier because of the complex and nonobvious assumptions currently made about the page control locking strategy.

A second item of interest is the management of the Bulk Store. It is currently planned that the last function of the core control program will be to make sure that there exist free paging device records and that the paging device map has been updated recently. This function is quite analogous to that of page stealing and is logically a completely separate task. However, due to the initial locking strategy (and the overhead of invoking the the core control task at all) it was thought that we might as well incorporate into it the paging device management as well. The final design would probably have the paging device map controlled by still another lock and the manager invoked at times independent of paging or core stealing.