TO: Distribution

FROM: Bernie Greenberg, Steve Webber

SUBJECT: Page Multilevel

DATE: 3/6/75

        The enclosed paper is being presented at the   IEEE   INTERCON
conference   in   New   York in April. This MTB is a preprint of the
paper for interested readers.

THE MULTICS MULTILEVEL PAGING HIERARCHY

Bernard S. Greenberg and Steven H. Webber
Honeywell Information Systems Inc.
575 Technology Square
Cambridge, Massachusetts 02139

INTRODUCTION

This paper describes the Multics multilevel paging system, the Page Multilevel algorithm or PML for short, with particular emphasis on the algorithms used to move pages from one level of the storage hierarchy to another. The paper also discusses some of the history and background of the development in particular where it relates to changes in the algorithms.

Although Multics has been in working existence for many years, many of its features are still novel and implemented on few if any other operating systems. For this reason, a discussion of some of the terminology as it relates to Multics is also included as background for the reader. Finally, a discussion is presented which predicts probable future developments both on Multics and other systems with respect to hierarchically organized memories (storage hierarchies) in light of what we have learned from Multics.

BACKGROUND AND TERMINOLOGY

Multics (C2) is a demand-paged system in which all addressable information is divided into 1024-word blocks, or pages, that are automatically read into main memory when referenced, if not already there. A page referenced that is not in main memory causes a page fault. The occurrence of a page fault is, of necessity, transparent to the executing program (although a real-time delay may be observed). The division of information into pages is likewise transparent to programs -- dynamic address translation hardware effects both the page mapping and the detection of page faults (B1).

The software that oversees paging activities, resolves page faults, and manages the paging data structures is known as page control.

In a demand-paged system, the resolution of a page fault usually requires the eviction of some page of data from main memory to make room for the page that has been faulted upon. The choice of which page to evict is crucial since the minimization of page faults (and, hence, paging overhead) is a function of

success in evicting those pages least likely to be used in the
near future. (The optimal replacement algorithm can be shown to
be that which evicts the page that will not be needed for the
longest time of any of those currently in main memory.) This
requires some prediction of program behavior.

The Multics main memory management algorithm uses an
approximation of the Least-Recently-Used (LRU) algorithm to model
program paging behavior (M1). The model implied by this
algorithm maintains that pages not used recently are less likely
to be needed than those used recently.

Other than pages in use as peripheral I/O buffers, any data
page may occupy any page frame of main memory. A page frame of
secondary storage is known as a record. For the most part, the
paging scheme does not recognize "ownership" of pages, or
recognize any special attributes of the data objects of which
given pages are a part. All pages are equivalent with respect to
replacement. In the current scheme, pages faulted upon by one
user can replace pages faulted upon by himself as well as others
-- the various management algorithms view usage only in the
global context of the whole system.

Thus, pages form a single large collection of data objects,
moving among levels of the paging hierarchy, each level forming a
single pool of page frames, without regard to "ownership" of
pages.

Multics employs a segmented virtual memory. The term
virtual memory has been used in the past few years to denote at
least two distinct, although not altogether unrelated concepts.
The first is roughly the concept of demand paging. A user
program "sees" an address space potentially "larger" than the
size of physical main memory. The second denotes a file system,
or similar data-object management system, in which data objects
are directly addressable by hardware, i.e., the file system is
part of the "virtual address space" of running programs.
Multics's use of the term virtual memory resolves this ambiguity
by conforming to both definitions (B1). Multics maintains a
directory hierarchy, a hierarchically organized collection of
segments, that are either directories (non-terminal nodes) or
non-directory segments (terminal nodes). Note that the (logical)
directory hierarchy should not be identified with or confused
with the (physical) storage hierarchy mentioned above.

Segments are data objects, each consisting of an array of
words. Programs, data bases, work areas, and the various
components of the supervisor itself, are all segments.
Directories are segments as well. They are catalogues,
containing branches for the segments and directories inferior to
them. A branch describes the names, access rights, and creation
and usage information for the segment or directory it describes,
and whether it is, in fact, a directory or a terminal segment. A

branch also includes a record of the physical location of the segment. A segment is transparently divided into pages, each of which has a _home_ on disk. The branch gives the disk address of the home of each page.

A Multics _process_ can be defined as an _address space_ and an execution point therein. An address space is a dynamically variable subset of the directory hierarchy constrained by the access rights of the associated process. (A process is roughly analogous to and usually associated with a logged-in _user_.)

Any data addressable by user or supervisor programs is conceptually in some segment. All CPU instructions develop two numbers, a _segment number_ and a _word number_, for each memory reference. The dynamic address translation (_appending_) hardware of the CPU maps the segment number, through a per-process mapping table (the _descriptor segment_) into a _page table_. This page table is uniquely bound to some segment in the directory hierarchy. Its function is to map all references made by that user using that segment number (or, significantly, any other user using some other segment number for this segment) into main memory addresses, if possible, or page faults, if not. The identification of this page table with a particular segment in the directory hierarchy provides for the resolution of a page fault by reading the corresponding page of the segment into main memory. Thus, the illusion of addressing "files" is maintained by these disciplines.

Segments may have their pages scattered over a disk storage subsystem: within each disk subsystem, one pool of unused disk pages is maintained. No attempt to perform input/output of contiguous pages is made. This is a result of the aforementioned equivalence of pages.

The large number of segments and the limited space available for page tables (which must be resident in main memory) necessitate a page-table multiplexing strategy. Another pseudo-LRU scheme is used to _deactivate_ segments when a page table is needed. Deactivation consists of removing the segment from the address space of all processes whose descriptor segments reference its page table, irreversibly flushing all of its pages from main memory, updating usage information in its branch, and finally, freeing its page table, and marking in its branch that it now has no page table. Subsequent reference by any process via the segment number which had described the segment in that process causes a _segment fault_ to occur in that process. The hardware, being asked to use the specially-marked entry in the descriptor segment of that process, produces this type of exception. The resolution of the segment fault consists of determining the identity of the segment from the supervisor data bases of that process, inspecting its branch to see if it is _active_ (has a page table), _activating_ it (creating a page table by deactivating some other segment) if not, and making a

descriptor segment entry to point to that page table. The software responsible for activating and deactivating segments is known as segment control.

This activation/deactivation strategy causes problems for the paging scheme. There exist special entries in page control for evicting all pages of a segment and page table replacement. Page control must be cognizant of segments being deactivated, and aware that a page table may denote many segments in succession in time.

PML partitions storage into a cost/size/access-time paging hierarchy. It is based upon the use of three storage media (in the current system fast main (core) memory, slower core memory (originally "drum"), and movable head disk) of decreasing average access speed and decreasing average cost per bit. It was by and large motivated by experience with Multics on the Honeywell 645 (GE-645), which employed a drum and a moving head disk subsystem (the drum, in actuality, was a very high speed six-platter fixed-head disk).

Advantage was taken of the speed and hardware queuing characteristics of the drum. As processes were suspended from active execution in main memory, as demanded by the job scheduling algorithm, drum-resident pages which were "attributed" to the departing process by several heuristics were written out en masse to the drum, in order to increase the ease of replacing these pages in main memory. This technique, post-purging, took advantage of the knowledge that those pages had an extremely small likelihood of being used while the departing process was not active in main memory.

Another technique dependent upon the drum's performance was pre-paging, which attempted to avoid page-fault overhead by reading in a subset of those pages which were post-purged when a process resumed active execution in main memory. The assumption here was that parallel issuance of I/O requests for these pages, with their identity already known, is more efficient than sequential faulting via demand paging, the latter incurring substantially more overhead and real time.

Although pre-paging and post-purging may bring to mind "swapping" on earlier systems, the intent was quite different. In their original conception and implementation, they were both schemes to take advantage of the large drum channel capacity and of position sensitive hardware queuing by the drum hardware and software. Were pages on disk post-purged or pre-paged, the resulting demands upon the disk channels would have been unacceptable.

HISTORY

As mentioned earlier, Multics in its early years made use of a drum. This drum had a transfer rate of one page in 2.1 milliseconds and an average latency of around 18 milliseconds. The first use of the drum was as a normal secondary storage device with no special treatment by page control. It was soon learned that the drum's speed could be better utilized if those segments which resulted in most paging activity were placed on it. This was done by determining the device on which a segment would reside when that segment was created.

Static Device Assignment

To better use the drum a program was called at segment creation time to determine through somewhat limited and artificial, but effective, heuristics exactly which of the secondary storage devices the segment should reside on for its lifetime. Since the drum was limited in size (4,096 pages) it was important to make a good decision at segment creation time. Unfortunately, the only information available at segment creation time are items such as:

1. location in directory hierarchy
2. directory/segment status
3. per-user status

Note that it has been a Multics policy not to accept user inputs on matters such as these; more consistency and dependability result if the operating system itself attempts to set and determine parameters related to performance and tuning. Hence, there was not and still is not any means available for a user to indicate what kind and degree of use a segment (or page) will get.

The early algorithm was basically as follows:

1. if the segment is a directory place it on the drum,
2. if the segment is a temporary per-process segment (and, hence, contains short-lived information) place it on the drum,
3. if the segment is in one of the system libraries (and, hence, probably used a good deal as well as being shared) place it on the drum,
4. otherwise, place it on one of the slower disk subsystems.

Note that decisions 2. and 3. above can be made solely from the segment's position in the directory hierarchy. Also, it should be pointed out that the appropriate overflow tests were made and that a segment would not be placed on the drum if there were not enough room, even if it passed the heuristic tests.

The final device assignment algorithm used before PML was installed placed the following segments on the drum:

1. supervisor segments
2. system library segments
3. the first 9 segments created in a user's process (temporary per-user) directory.

This scheme had several problems. First, it placed system files which may not be used at all on the drum. Mere presence in a system library does not guarantee use, let alone heavy use. Second, the scheme placed a segment entirely on the drum or not at all. This meant that a segment with a few heavily used pages might reside on the drum, thereby wasting critical drum records for the unused pages of the segment. Third, segments grow after they are created and often after they have been around for a while. The test to determine if there were enough drum records available was not deterministic and drum overflows could (and did) result. (The "obvious" choice of using the sum of the maximum lengths of all segments on the drum as an overflow criteria would have wasted most of the drum.)

Of these flaws the first was the worst. By placing little-used segments on the drum it wasted drum space and prevented other heavily used segments from being placed on the drum. For this reason, the following extension was incorporated into the system.

Segment Migration

Segment migration is the term given to the technique used to move segments from one secondary storage device to another as a function of recent (but not transient) use. The use of a segment was determined by the number of page faults incurred by the segment per unit time. The method to effect the movement of segments to their most appropriate device was as follows: first a pass was made over the entire directory hierarchy to determine the use of each segment and select the most heavily used for the fastest device (the drum), the next most heavily used for the next fastest device, and so on. After the first pass, came a second pass which

1. reset the usage statistics, and
2. moved each selected segment, page by page, to its new device.

Although a criterion is still needed for determining the initial device for a segment, the criterion now could be more conservative about placing segments on the drum since highly used segments will migrate there.

The actual segment migration algorithm was amended to look at several attributes of a segment to determine the final device assignment. In particular, segments not used in, say, N days

would not be placed on the drum. Also, segments created within,
say, H hours would not be placed on the drum (this eliminated
some problems with transients).

     The program to "adjust" the contents of the devices (place
segments on their appropriate storage device) was run weekly.
Hence, any mistake in the algorithm could be corrected the next
week. Although it was not done so, it would have been easy to
perform the adjustment automatically with no external initiation
at any specified intervals.

     The actual system overhead of this scheme was about 2 hours
of system down time per week. This was less than 2 percent of
the system availability. This scheme was used for over a year
and although it was better than the static device assignment, it
still could not solve the last two problems mentioned in the
earlier section, namely unused pages of a heavily used segment
would be placed on the drum, and drum overflow could still occur.

OBJECTIVES

     Up to now we have been assuming the truth of the statement
that minimizing the time to get a page into main memory after a
page fault would improve system performance.   What we really
would like to do is minimize the weighted average page wait delay
(weighted by percentage of faults from each device) for all
devices. We do this if we can force the system to take as many
of its page faults as possible from the fastest device. This is
why the various attempts were made to get pages onto the drum.

     There are, however, other strong reasons why minimizing the
average page wait delay is advantageous. User response time,
system efficiency, and cost effectiveness are all improved if we
do.   The user response time on a paging system such as Multics
must include real time delays to re-establish the set of pages in
main memory necessary for useful work to proceed, i.e., the
user's _working_ _set_ (D1). The more pages in the working set and
the more of these not on a fast device the slower will be the
user's response time.   Pre-paging (mentioned earlier) can
decrease this delay but only pages on a device such as the drum
can reasonably be pre-paged.

     The system efficiency can be increased as well. While a
process is waiting for a page, it can not proceed.   While it
remains dormant another process is usually run which itself will
likely cause a page fault. Any such page fault by other
processes has the potential of forcing one of the initial
process's pages from main memory -- resulting in potential
thrashing. Such behavior, although possibly controlled by main
memory partitioning in some manner, is a prime cause of overhead
in a demand-paged system such as Multics. By minimizing the page
wait delay the likelihood of having one of the process's pages
claimed by another process is likewise minimized.

By increasing system efficiency the cost effectiveness of the system -- with the same hardware -- is improved. An interesting question, then, is what hardware is required to achieve the same system efficiency and user response given a new algorithm. We will touch on this briefly later on.

So, to decrease the average page wait delay and to solve the problems mentioned earlier (unused pages on the drum and drum overflow) the page multilevel algorithm described in the next section was devised.

THE SOLUTION - PAGE MULTILEVEL

Until PML was introduced into Multics, a page was either in main memory or it was not. If it was not in main memory (and it was non-null) it resided on some secondary storage record. If it was in main memory, the secondary storage copy, if it existed, was not necessarily up to date. There are 3 valid states of a page in this scheme:

1. in main memory - disk copy up to date

2. in main memory - disk copy old or not yet created

3. not in main memory - disk copy up to date

PML adds a third level to this memory structure. A page can be in main memory, on the paging device, on disk, or some combination thereof. The actual legal combinations is a function of the algorithms used.

The advantages of such a scheme are obvious. The most recently used pages would be left on this intermediary paging device. They would be retrieved from the paging device at page fault time rather than from disk and hence the average page wait delay would be reduced. The use of such a scheme relies completely on the LRU algorithm and the associated behavior necessary for such an algorithm to succeed.

By organizing the scheme on a per-page basis the earlier problem of wasting drum records is avoided. By having appropriate fallback options when a record of the paging device is needed but none are available, the problem of drum overflow is solved. By making appropriate choices as to when to place a page on the paging device, the system reacts much faster to changes in the entire set of pages used by all users on the system.

The Choice of an Algorithm

There are three basic decisions which must be made when designing a page multilevel algorithm such as this. These are:

1. How and when do we update the secondary storage disk,

2. How and when do we update the paging device, and

3. What type of replacement algorithm is to be used for the paging device.

     None of the questions has an obvious answer, but we will describe why we made the choices we did.

Updating Secondary Storage

     We considered three obvious options for updating secondary storage. These were specifically:

1. Update secondary storage whenever the paging device is updated (store through),

2. Update secondary storage whenever the paging device is updated but only for certain classes of pages (conditional store through),

3. Don't update secondary storage until the paging device copy is being deleted from the paging device to make room for a more heavily used page.

     The advantage to option 1. is that no mechanism need be created to move pages from the paging device to secondary storage when freeing a paging device record. Also, secondary storage would always be up to date so that little would be lost if a system failure were to occur and the contents of the paging device were unrecoverable. The main disadvantage with this scheme, and the reason it was not chosen, is that it would cause an unnecessarily high amount of traffic to the secondary storage devices. In particular, so many write requests would be queued that the page wait delays attributable to secondary storage I/O queueing would almost always be significant as the system waited for a write to complete before queueing any reads.

     The advantage to option 2. over option 1. is that the amount of secondary storage traffic can be controlled by limiting the types of pages written immediately to secondary storage. The disadvantage here is that a mechanism must be provided for moving pages from the paging device to secondary storage for use by the paging device replacement algorithm.

     The advantage to option 3. is that still less secondary storage traffic results. However, the need for a function to move pages back to secondary storage still exists.

     Multics actually implements a combination of options 2. and 3. whereby the type and number of pages written immediately to

secondary storage is controlled administratively. The options
are:

1.   Store through all (non-temporary) pages,

2.   Store through only directory pages, and

3.   Store through no pages.

     Since it is not guaranteed that all pages will be written
automatically to secondary storage, the function of moving a page
from the paging device to secondary storage had to be provided.
This function, known as a read-write sequence is described later.

     As will be discussed later, it is not necessary for a
read-write sequence to be performed each time a record of the
paging device is needed.

Updating the Paging Device

     The following options for moving pages to the paging device
were considered:

1.   Write a page to the paging device if it is not there but must
     be evicted from main memory.

2.   Write a page to the paging device if it is not there, must be
     evicted from main memory, and has already been evicted from
     main memory once since it was activated.

3.   Write a page to the paging device if it is modified and must
     be evicted from main memory, and

4.   Do not write a page to the paging device if it belongs to a
     certain class of segments which are never to be written to
     the paging device.

     All of these options are available to Multics with the
appropriate setting of administrative parameters.   A fifth
option, that of dynamically monitoring the use of a page over a
period of time and using this information to determine when to
write it to the paging device was rejected because it was too
costly to implement and because it did not react in time to be
fully effective. (Again, program behavior consistent with the
LRU algorithm is assumed.)

     The standard mode of operation is that described by Option
1. Whenever a page is evicted from main memory, it is moved to
the paging device if it is not there. This subsumes Option 3.
Option 2. can be set by administrative control and is intended
to be used when the system is running many programs that
reference pages exactly once and not again for a long time.   (If
there is only one reference nothing is gained by moving the page

to the paging device.) This type of program behavior might appear when long, sequential searches or data copying are frequent. (Any pages really used repetitively will be moved to the paging device on the second reference under this scheme.)

A feature was added to the system to allow all segments referenced by only one process (in the Multics sense) to be conditionally moved to the paging device. This again, would be used by processes knowing their behavior, a priori, and is of limited use. One such process is the Multics backup dumping process which copies data from disk to tape and typically touches each page only once.

The Replacement Algorithm

There were several choices before us when the replacement algorithm was undertaken. These were reduced to the following:

1. Evict pages from the paging device as needed, i.e., when a paging device record is needed.
2. Keep a free pool of paging device records to be used when needed.

The advantage to the first scheme is that it does not waste paging device records in the free pool as in the second scheme. The disadvantage with the first scheme is rooted in the heart of the Multics main memory replacement algorithm. When a page fault occurs (and in certain other cases), a program is called that is to return a frame of main memory that can be used as the caller sees fit. This same program must be called by the paging device replacement algorithm to get a frame of main memory for a read-write sequence. Further it is this same program that needs to force a page to the paging device as part of the general PML scheme. It can readily be seen that an awkward recursion problem arises if the paging device eviction is done at main memory page eviction time. To avoid this recursion and the associated bugs, overhead and complexity, the second alternative of managing a pool of free paging device records was chosen. This pool could be replenished at more convenient times. Indeed, when to replenish this pool is an interesting decision. It was decided to replenish the pool when the number of records in the pool fell below a given threshold and that this threshold is checked just prior to handling each page fault, but before any potentially recursive code is entered.

The Implementation of PML

PML was implemented in late 1971 in the context of the page control subsystem as it existed at that time. Thus, many of the primitives were extensions to existing primitives, while the others relied heavily upon them.

The management of the paging device is carried out by means of the <u>paging device map</u> (PDMAP), a main-memory resident map of the paging device, with an entry for each record of it. Each entry which corresponds to an in-use record contains the corresponding disk address, a pointer to the page-table entry corresponding to this page (if and only if it belongs to an active segment), and various status flags. The map is indexed by paging device address.

The PDMAP entries are kept in a linear <u>paging device list</u>, as well. One end of the list contains all free entries. The rest of the list implements an LRU discipline on the paging device, with the end not having the free entries being most recently used. Hence, replenishment consists of converting least recently used entries into free entries.

The LRU discipline is maintained on the assumption that main memory is substantially smaller than the paging device. Hence, any page that is in main memory, if it appears on the paging device, must be among the most recently used on the paging device. Thus, PDMAP entries are threaded to the most-recently-used list position upon:

1. page fault on that page -- this is actual use.

2. discovery of this page in main memory by the paging device replacement code.

3. eviction of this page from main memory.

Paging device space is allocated when a page in main memory must be replaced, and no copy of it exists on the paging device. At this time, a record of paging device is allocated (removed from the free list), and the main memory page written out to it, regardless of whether or not the page is identical to its disk copy. (Pages are never written to a record to which they are known to be identical.) Hence, the next fault referencing this page may be resolved from the paging device.

At the time that paging device space is allocated, free paging device records must exist. If none exist, none can be allocated. None can be created at this time, because paging device replenishment invokes read-write sequences, which themselves require pages of main memory, and the previously mentioned recursion problem is created. Therefore, one of the first duties of the page-fault handling routine is to insure that a fixed number of free paging device records are available, in anticipation of need for paging device space during the search for main memory for resolution of the page fault.

Replenishment of the paging device consists of freeing enough records to make a fixed number of free records. The records freed are the least recently used records on the paging

device.   Freeing a given record consists of insuring that the
disk copy, which always exists, is not different from the paging
device copy.   A flag is maintained in the PDMAP entry, which is
set ON whenever the corresponding page is noticed to have been
modified while it was in main memory. If this flag is OFF, the
record may be freed immediately, as the paging device copy must
be identical to the disk copy.   If it is ON, a read-write
sequence (RWS) must be performed, updating the paging device copy
to disk, for the paging device copy contains modifications not
present in the disk copy.

     Performing a read-write sequence consists of four steps:

1.   Finding or creating a free frame of main memory.   This may
     include allocating (using up a free) paging device entry, but
     a critical assumption here is that this is unlikely, as most
     of the pages of main memory will have paging device copies.

2.   Reading the paging device frame into the main memory frame.
     With the current slow-core paging device, this is done as an
     indivisible operation of page control, i.e., no change in the
     state of the page of data or frame of main memory can occur
     while it is in progress.

3.   Starting a write to disk. The actual disk operation can take
     some time, and hence this cannot be treated as an indivisible
     (protected) operation.   Thus, while this write is   in
     progress, attempts may be made to deconfigure (remove) the
     main memory frame being used as a buffer, destroy the page of
     data, or fault upon it.

          While memory deconfiguration will await the completion
     of the RWS, an attempt to fault on the page will cause a
     special event known as an RWS abort to occur. This causes
     step 4. below not to free the paging device record, but
     instead, reinstate it and resolve the page fault (using the
     frame of main memory used by the RWS).

          An attempt to destroy the page causes an RWS truncate to
     occur. This causes step 4. below to take special action.

4.   Freeing the paging device record.   Or, if an abort has
     occurred, rethreading the paging device map entry to the
     most-recently-used end of the paging device list   and
     connecting the page table to the buffer frame.

     Of the above four steps, steps 1. through 3. are performed
at the time replenishment of the paging device is sought.   Step
4. is performed at interrupt time.

     The store through option is implemented as a flag set at the
time a paging device write is started. At the (interrupt) time
this write completes, the flag, if ON, is reset, and a disk write

started.

     In the Multics page control subsystem, interrupt time
operations happen either at the time of a physical device
interrupt, or "on demand", when a traffic bottleneck is detected.
This latter operation, known as _running_, polls all device control
routines for completed operations.   As masking inhibits
interrupts, and a lock excludes other processors during paging
operations, this is the only way to learn of operations completed
at these times. Running is performed when an excess of writes
(30) have been queued by main-memory management in search of a
free frame, and by paging device replenishment when an excess of
read-write sequences are in progress. Device control routines
often "run" themselves to free queue space, representing
completed operations, when called to initiate an operation.
Interesting problems of recursion and asynchrony are posed here.

     An auxiliary data base used by the segment control subsystem
in interfacing to PML is the _paging device hash table_. This
table is necessitated by the policies of keeping pages of
segments which are not active (have no page table) on the paging
device, and not marking the permanent branches of these segments
as to if and where on the paging device these pages might exist.
Hence, the paging device hash table maintains a mapping from disk
address into paging device map entries. This table is organized
as a table of equivalence classes, being selected by the low bits
of the disk address. The most recently allocated PDMAP entry in
this class is selected, and it is the first of a list of entries
in this class. At the time a segment is activated, all disk
addresses in it must be hashed into the PDMAP so that the correct
paging device address may be placed in the page table being
created if the page exists on the paging device. Also, if a page
of a non-active segment is destroyed, any paging device map entry
which may exist must be freed. The paging device hash table
serves this need as well. Although the strategy of keeping pages
of non-active segments on the paging device may appear
unwarrented since only a very small number of such pages are
usually found on the paging device, experiments indicate that the
necessary hashing is an almost unmeasurably small fraction of
system overhead.

RELIABILITY

     A good deal of design effort was spent to ensure that use of
the paging device would not be a cause of unreliability. In
particular, since it was decided not to use a complete store
through algorithm, the most up to date (or only) copies of
segments (and directories) might well be only on the paging
device. If the system were to fail such that we could not update
all the information to secondary storage, we must be able to
recover as much as possible from main memory and the paging
device itself. The critical data base here is the paging device
map which maps the data on the paging device to its home address

on secondary storage. This map is in main memory and may be lost
if a power failure destroys the contents of main memory. Since
the map is so critical it was decided to frequently write it onto
the paging device itself so that if the contents of main memory
are lost the mapping will not be. The map is currently written
out once each second.

As a further check against loss caused by a system failure,
the option of using store through (at the expense of secondary
storage channel queueing) was provided. This is used primarily
for directories at some Multics sites but can be used for all
pages if desired.

Another safety check against the loss of the contents of the
paging device itself is the requirement that certain critical
segments not be allowed to be placed there. The free storage
maps for secondary storage and the directory hierarchy root are
two such segments.

Extensive software for automatically (and manually)
deconfiguring faulty (unreadable) records of the paging device
was added.

EVALUATION

PML meets its objectives as already defined. On a Multics
system, using 2.0 microsecond core memory as a paging device
(2048 pages for 300 pages of main memory), from 85% to 90% of all
page faults occur on pages already on the paging device.

The disadvantages of the implementation are several. Many
of these problems point out design tradeoffs, or are reflections
of larger design decisions in Multics as a whole.

The need for read-write sequences introduces paging-related
overhead. An estimated 1.2% of system CPU time is spent in this
activity. Read-write sequences require main memory, and the
allocation of main memory requires writing out pages. Writing
out pages may require the allocation of paging device records,
which requires the freeing of the same. As this freeing involves
read-write sequences, a recursion is created, whose full solution
involves an unacceptable overhead in terms of Multics page
control. Hence, recursion escapes and heuristics are involved in
place of this recursion.

PML complicates the main memory replacement algorithm
substantially, as pages in main memory which are identical to
their disk copies must be written to the paging device when being
evicted from main memory.

The decision to write a page in this manner is somewhat
complex and thus, the replacement algorithm is slowed finding a
main memory frame with which to resolve a page fault.

The policy of keeping pages of deactivated segments on the paging device is questionable at this point. Less than one percent of the pages on the paging device represent deactivated segments. The need for keeping track of these pages necessitates the paging device hash table, and the complexity associated with it. Were such pages simply flushed from the paging device at the time that these segments were deactivated, there would be no need for this data base.

All of the considerations so far have contributed to the complexity of the page control subsystem: the effectiveness of the Multics virtual memory is inversely related to the time spent in paging overhead. What is more, the high degree of asynchrony employed in the page control subsystem causes problems in this area to be particularly difficult to diagnose, and often irreproducible. Hence, complexity is a larger issue in this area than in most other areas of the operating system.

METERING

The Multics system has extensive software meters throughout the supervisor and the page multilevel algorithm is no exception. Records (counts) are kept for all normal and abnormal events including average retention period for pages on the paging devices. The cost of the entire page multilevel system is recorded. Some typical values for a typical system are:

#Page faults	3111214
Average time on PD	10 minutes
% faults from PD	85%
% writes forced (recursion escape)	.03
# RWS	71621
% PD records modified	30%
% PD records free	2%
% PD records active	99%
% system overhead	2.1%

These figures are from a 384K main memory, 2 processor, 2 million word paging device over an eight-hour period with from 40 to 70 users logged in.

Another meter of interest is a histogram that shows how recently used each page faulted on is by noting where the map entry for the page occurs in the paging device list of the PDMAP. Discussions of the results of this meter can be found in (G1) and (S1).

The performance of the system as a whole has also been measured with different sizes for the paging device. A certain script of 40 jobs will run on a system (256K main memory, 1 processor) with 2 million words of bulk store in about 60 minutes. The same script with only 256K of bulk store will run

in over 200 minutes.

    When Multics converted the drum from a segment migrated
secondary storage device to a paging device (same amount and kind
of storage, just used in a different way) a 20% to 25% increase
in useful work done by the system was noticed.

FUTURE

    The future of PML in its present form is unclear. As memory
technology makes main memory cheaper, vast amounts of main memory
will reduce page fault rates to an extent where resolution of all
page faults from disk may be acceptable. Were the two million
words of slow core memory being used as a paging device
addressable as main memory, none of the overhead of transfers and
read-write sequences would be necessary. Processor cache
technology can increase the effective processor store access rate
to the point where "slower" main memory, with a cache, is
measurably faster than the "faster" main memory alone.

    The concept of a multilevel storage hierarchy is not
obsolete: cache is the extension of main storage on the faster
side; on the slower side, any of various forms of automatic and
user or administratively invoked migration schemes can be
envisioned. Tape backup forms a part of such a hierarchy.

    Faster disks are creating interest in techniques akin to
"swapping" in older time-sharing systems and pre-page/post-purge
in Multics. A scheme is being investigated where working sets
may be "swapped" in and out using scatter/gather I/O to a single
high-speed disk track. Although this scheme combines all of the
advantages of pre-paging (reduction of fault overhead by
anticipating need for pages), post-purging (early freeing of main
memory with little likelihood of use), and wholesale swapping
(maximum possible transfer rate to and from disk), the problem of
mapping the swap images into the hierarchy seems prohibitively
complex at this time if coordinated within the PML scheme.

SUMMARY

    The Multics paging subsystem has been successfully extended
to use high performance memories in a paging hierarchy to
substantially increase the throughput and response of the system.
Advances in memory technology forcing adjustment of pricing
schemes and related price/performance values are leading us to
study new techniques for hierarchically organizing the available
storage devices including scatter/gather swapping and processor
cache hardware. The value of hierarchically organizing the data
is apparent. The actual mechanisms used will be the subject of
many future studies.

ACKNOWLEDGEMENTS

BIBLIOGRAPHY

B1   Bensoussan, A., Clingen, C.T., and Daley, R.C., "The Multics
     Virtual Memory: Concepts and Design", Communications of the
     ACM 15, 5 (May, 1972), pp. 308-318.

C1   Corbató, F.J., "A Paging Experiment with the Multics System"
     in Ingard, In Honor of P.M. Morse, M.I.T. Press, Cambridge,
     Mass., (1969), pp. 217-228.

C2   Corbató, F.J., and Vyssotsky, V.A., "Introduction and
     Overview of the Multics System" Proc. AFIPS 1965 FJCC, Vol
     27, Pt. 1. Spartan Books, New York, pp. 185-196.

D1   Denning, P.J. "The Working Set Model for Program Behavior",
     Communications of the ACM 11, 5 (May 1968), pp. 323-333.

G1   Greenberg, B.S., "An Experimental Analysis of Program
     Reference Patterns in the Multics Virtual Memory", Master's
     Thesis, M.I.T. Dept. of Electrical Engineering, May, 1974.

M1   Mattson, R.L., et al., "Evaluation Techniques for Storage
     Hierarchies", IBM Systems Journal 9, 2 (1970), pp. 78-117.

S1   Saltzer, Jerome H., "A Simple Linear Model of Demand Paging
     Performance", Communications of the ACM 17, 4 (April 1974),
     pp. 181-186.