Multics THROUGH THE LOOKING GLASS

A Paper Presented to the HLSUA Forum XXXI October 12, 1980

Tom Oke
Senior Computer Systems Analyst
Academic Computing Services
The University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada T2N 1N4

Abstract

This paper deals with some of the problems encountered at **The University of Calgary** during the tuning and optimization of system performance. It presents some of the characteristics to be found in both the scheduling system and the virtual memory environment of **Multics,** and attempts to put forward a heuristic model of system action to permit a tuner to improve performance.

System Experience with Multics

Multics arrived at **The University of Calgary** in the winter of 1978/1979, and was released to the users in the month of June for testing, taking over from a **CDC Cyber 172** in the month of July.

Since that time we have experienced both joy and sorrow, as with any new child, and have attempted to analyze and improve the system and our operation. During much of our **Multics** contact we have been running at and beyond any comfortable capacity limits and have been forced into some detailed analysis to improve our situation. Tools have been developed, as well as a possible model of operation of the **Multics** system, and are the topic of this paper.

The University of Calgary configuration used in this discussion consists of:

2	CPU (with cache)
1	SCU (1 million words)
1	SCU (2 million words)
1	IOM
2	MPC (dual channel, disk controllers)
12	MSU451 disk drives

Caveat

Of necessity this paper presumes some prior knowledge of Multics, configuration, and metering and tuning. It is hoped that the reader has had some experience in this area. If not, an additional source of information can be found in course notes from the F80 course, 'Multics Process Management Analysis'.

General Outline

This paper is presented in three sections. The first deals with the Virtual Memory System of Multics and presents the disk system and disk tuning. This section outlines the hardware and software components of the disk system, how they function, and how they may be set up to get maximum throughput and redundancy.

The second section deals with the scheduling concepts of Multics and presents a memory access model, which can be useful in tuning and understanding the operation of Multics. This model basically separates the resources of processor and memory, and permits one to better estimate the requirements of both, and what may be necessary to scale operations up and/or down.

The third section outlines tools, which can be useful to a tuner to understand bottlenecks and discover areas of improvement and degradation. The tools section further indicates standard tools and **The University of Calgary** tools.

Some of the information presented in this paper is a condensation of previously developed documentation, such as disk tuning, and notes on scheduler operation. If further information is desired it can be obtained from the author at the address noted in this paper.

DISK TUNING

The Multics virtual memory system depends upon the ability of the disk system to store data and rapidly service transfer requests. The disk system physically consists of disk drives, Micro-Programmed-Controllers (MPC's), and the Input/Output Multiplexor (IOM). **Multics** system software breaks this equipment into one or more disk sub-systems and controls the management of disk resources.

Disk System Hardware

Disk drives are connected to the MPC's through a Channel Adapter (CA). Each drive can be connected to two CA's, usually from different MPC's, to achieve redundancy and dual access. Each MPC typically handles up to 16 drives (expansion is possible to 32), and the MPC manages all seek overlap and control of IO transfers. The MPC connects to the IOM through Link Adaptors (LA's), which are sometimes termed 'Physical Paths'. An MPC may have up to two LA's, each of which effectively has its own processor.

Thus a dual LA MPC consists of two separate and distinct disk controllers sharing common disk connections.

The IOM accepts the physical path (a **PSIA board** in **the IOM chassis)** and logically divides the PSIA into a maximum of 8 logical channels. The number of channels, and the start channel for each PSIA, is set by resistor packs soldered on the PSIA board.

Thus a single physical path becomes one to eight logical paths at this point. Logical paths must be based on power-of-two channel boundaries, dependent upon the number of logical channels configured on the physical path:

Number of Logicals	Boundary
1	1
2	2
3-4	4
5-8	10(8)

Each logical channel is mapped into a specific main memory address termed a **MAILBOX,** which holds a complete request, either a read or write, and is tied up until request completion, by a disk drive, 'frees' the mailbox.

Seek Overlap Restrictions

Since there can only be as many active requests as there are logical channels, this limits the number of seeks which can be done simultaneously **(seek overlap).** This limit also governs optimization of IO since, if there are too few logical channels, some drive(s) must complete an operation before another request can even be issued. Running the drives serially, as done with too few channels, is obviously less efficient than running the drives in parallel, as having sufficient seek overlap does.

Disk System Redundancy - Physical

The path for a disk operation starts in the IOM with the placing of a request in a particular logical channel mailbox. The IOM passes this request to an MPC when an IOM interrupt arrives on the physical channel, which 'owns' the mailbox. The MPC receives the request on a particular LA and passes the necessary seek request to the appropriate drive through the cross-barred CA's of the MPC. The disk drive signals a completion to the MPC at the completion of the seek, and the MPC then requests a read or write. When the IO operation is complete, the MPC signals a completion to the IOM over the 'owning' physical channel, and the IOM signals an interrupt that ends up in 'dctl' in 'page_control_'.

As you can **see,** there is a single physical path from a **'mailbox'** to a disk drive, which consists of a physical path to the LA, an MPC half, and a common CA to the disk drive. One generally attempts as much redundancy as possible by splitting the LA's of an MPC across two IOM's, thus if one IOM dies, the other IOM can still carry traffic for the MPC's other LA. Drives are dual ported and are usually connected to two different MPC's, if an MPC dies then the other MPC can still carry traffic for the disk drive.

This physical redundancy does not take into full account the logical makeup of the disk system, since a single request can die with the IOM or the MPC and be lost forever, but it does assist in trying to bring the system up again, even if equipment is still down, without having to alter the CONFIG deck.

Disk System Logical Configuration - Logical Redundancy

Logical channels are assigned to disk sub-systems through PRPH and CHNL cards in the **CONFIG DECK.** These cards designate the disk drive makeup of a sub-system, the primary channel(s) to access the sub-system (PRPH), and the auxiliary channel set(s) to access the sub-system (CHNL). One can specify one channel set on the PRPH card, and up to three channel sets on the CHNL card (though a bug in 'interrupt_meters' prevents you seeing the sub-system name for the last two channel sets of the CHNL card).

The physical makeup of the disk MPC's is defined through the MPC card(s) in the CONFIG DECK, these define which channels are attached to an MPC, and further define the IOM(s) to which the channels are connected. Thus the MPC card defines the physical paths of the disk system.

The PRPH and CHNL cards define logical splitting of the logical channels, and since there is a one-to-many mapping of physical to logical connections, these cards also define the physical paths and their order of access for the logical disk sub-systems.

One can take advantage of the splitting of physical and logical channel assignment in the CONFIG DECK to explicitly define the paths and redundancy of access, to disk sub-systems, through judicious assignment of channels in the PRPH and CHNL cards. What we have done is to assign channels so that each sub-system gets a unique primary access channel, so that each sub-system has its own primary physical path. At **The University of Calgary** we have the following channel assignments to MPC's:

Channel	**MPC/LA**	**Sub-system**
08	00	DSKA
09	00	DSKC
10	00	DSKB
11	00	DSKD

12	10	DSKD
13	10	DSKB
14	10	DSKC
15	10	DSKA
32	01	DSKB
33	01	DSKD
34	01	DSKA
35	01	DSKC
36	11	DSKC
37	11	DSKA
38	11	DSKD
39	11	DSKB

This configuration is defined through the following CONFIG DECK entries:

```
PRPH DSKA A 8.  1 451. 3
PRPH DSKB A 32. 1   0  3  451. 3
PRPH DSKC A 36. 1   0  6  451. 3
PRPH DSKD A 12. 1   0  9. 451. 3

CHNL DSKA A 34. 1 A 37. 1 A 15. 1
CHNL DSKB A 10. 1 A 13. 1 A 39. 1
CHNL DSKC A 14. 1 A  9. 1 A 35. 1
CHNL DSKD A 38. 1 A 33. 1 A 11. 1

ROOT DSKD 12. DSKA 1 DSKB 4
MPC A  8. 4 A 32. 4
MPC A 12. 4 A 36. 4
PART BOS DSKD 12.
PART DUMP DSKD 12.
PART LOG DSKD 12.
```

This specifies that DSKA will be accessed through channels 8., 37., 34., and 15.; DSKB will be accessed through 32., 13., 10., and 39.; DSKC will be accessed through 36., 9., 14., and 35., and DSKD will be accessed through 12., 33., 38., and 11.

The channels specified above are also accessed in the order specified, through the channel re-ordering of the routine **'disk_init'**, which is why the order of channels specified on the PRPH and CHNL cards seems to make little sense. This order of specification is necessary since channels are chosen by 'disk_init' in the order:

4 Channels - 0, 2, 1, 3
8 Channels - 0, 4, 2, 6, 1, 5, 3, 7

from the order specified through the PRPH and CHNL cards. Thus the order

8., 34., 37., 15., comes out as 8., 37., 34., 15., and accesses in the sequence: MPC 0,LA 0, MPC 1,LA 1, MPC 0,LA 1, MPC 1,LA 0.

This specification causes each sub-system to have its primary access on a separate physical path from all other sub-systems, and to contain all physical paths in the specification of channels to each sub-system. The order of access to these physical paths is also set to quickly change to an entirely different MPC, to minimize a complete MPC down degrading the system. This causes access to physical paths such that the most heavily used primary physical paths will be the least heavily used secondary physical paths. Another advantage is that each sub-system has available to it a full complement of physical paths to optimize physical transfer as much as possible. The stock method of allocation would allocate a complete physical path, with its four channels, to each sub-system. If any physical path dies, the entire sub-system is inaccessible; if a full MPC dies, one loses two sub-systems.

The allocation given above has one more possible level of seek overlap than it has drives, permitting one of four physical paths to die and still retain full seek overlap. In any case of degradation, down to having only one of four physical paths functional, this allocation permits the full disk system to be accessible and degradation is always evenly distributed.

Caveat

Use of channels over 32. is necessary for normal disk allocation, but the old software limits of GCOS, BOS, and T&D's can cause problems unless care is taken. To boot T&D's on such a system, the channel plugs for all channels greater than 31. must be pulled, and then replaced after T&D's are up. In addition channel allocation must ensure that the rpv is accessible, and bootable from a channel less than 32., otherwise BOS simply won't come up.

What You Tune For with Disk Sub-systems

If one uses the method outlined above to allocate disk channels and sub-systems, there is little that can be tuned for in the physical makeup of the disk system, other than attempting to even distribution of heavily hit drives across sub-systems. Physical accessibility of the system cannot be optimized any more and disappears from tuning consideration.

Disk Scheduling Discipline - Nearest Seek

What concerns a tuner, in the disk area, is optimization of the logical use of drives and controlling accesses to **Virtual Memory** through optimal scheduler tuning.

Logical access to disk drives is controlled by the routine **'dctl'** in

'page_control_'. This routine uses a **nearest seek** discipline to optimize disk accesses and maintains queue information for all drives of a sub-system in two queues, a high priority queue for **page reads** and **VTOCE IO,** and a low priority queue for **page writes.** Since the queues are split there is no way to fully optimize a combination of page 'reads', and page 'writes', this is seen frequently through allocation and run locks.

The Multics system can only sponsor a page read request for each eligible process, thus the total number of page reads which can be queued for the entire system is limited ('maxe' + the number of 'realtime' processes). Page writes are sponsored by a non-blocking cleanup mechanism, and there can conceivably be as many writes queuable as there are main memory frames. Typically twice as many reads as writes are done in the long term, but the request characteristics are such that reads occur steadily, while writes occur in bursts as old modified pages are cleaned up. This tends to produce a write queue, which always has something in it, and a read queue, which steadily gets requests. The priority separation between queues always terminates optimization of a queue of writes to service a higher priority VTOCE or read operation. With the complete separation preventing any cross-optimization this system is inefficient in times of high IO loading of the disk system, eliminating most of the optimization of both reads and writes. This can physically be seen by watching the disk head of a very active drive, with most seeks being long distances rather than the expected sliding action across the pack surface.

Queuing Problems in Disk Control

The above system characteristics produce queuing problems, particularly on small memory systems with low lap times, resulting in **run locks** and **allocation locks.** Run locks occur in two situations, when **'pxss'** sponsors a **'run*** of the devices every 15 seconds, and when the number of write requests currently queued for all sub-systems reaches the value set by the parameter **'write_limit'.** This value is either explicitly set with the **'ctp'** command, or through the PARM card WLIM parameter in the **CONFIG DECK,** i.e.,

```
    PARM TTYB 16384. WLIM 90.
```

If 'write_limit' is not explicitly set, it defaults to 1/8th of the number of available pages in the system, which is usually too large, since each sub-system can only hold 64 queued requests and there are usually fewer than 2 disk sub-systems per million words of main memory. The best idea for a systems tuner is to explicitly set 'write_limit'. If you see few **'run_locks'** showing up in **'disk_meters'** over a very busy period, or the number of sub-systems*64 is less than 'write limit', then you probably have a system with an unreachable 'write limit'.

Run locks entirely shut down system paging, to the extent that no page operations are done until one or more writes complete. This means no more writes are queued, no more reads are queued, and 'page_control_' will not even attempt to check if a 'free' page exists.

Allocation locks occur when more than 64 requests are queued up in a single disk sub-system, causing 'dctl' to wait until any IO operation completes before queuing the current request. This usually stops any paging activity, but has the slight advantage that if one of the more optimized VTOCE or page reads completes, the current request can now be queued. However, this has the disadvantage that the completion must occur on fewer drives than for a 'run_lock', reducing the probability of a completion and increasing the delay. In addition, allocation locks usually occur in the middle of the burst of 'writes' being queued by 'page control ', meaning that as soon as the current request is queued, another write will be tried and will cause another 'allocation_lock'. Thus, allocation locks are a long-term blight. If given a straight choice between an equal number of 'run locks' and 'allocation locks', the best choice would be 'run_locks', and 'write_limit' should be reduced, but this is rarely the choice and need not be the choice, since there is another way.

When to Increase the Number of Sub-systems

There are two basic reasons for increasing the number of sub-systems.

1. **You have more than eight drives on a single sub-system and can still configure more logical channels.** You can gain back full seek over-lap and better degradation characteristics by splitting sub-systems to have less than eight drives, but up to eight logical channels. Further thought should be given to the actual channels, their sources and the order in which they will be used to gain maximum channel use and path redundancy. REMEMBER, you can only put up to eight-level seek overlap on any sub-system since it is limited to a maximum of eight logical channels. If you need more overlap, the only way to get it is to split sub-systems. The best way to determine if you need more overlap is through the output of the 'disk_queue' command, or the locally written 'find' command (The University of Calgary). But overkill is the easiest way to handle degradation situations.

2. **You are incurring too many allocation locks**. Unless these are mainly from a single drive you can handle the situation by splitting sub-systems and splitting busy drives across the storage resources of the pool of queue entries found in each sub-system (64 entries per sub-system). You should also think strongly about reducing time quantums to buckshot (scatter) the pages of heavy hit segments more evenly across the core map.

If you are incurring too many 'run_locks' then you need to increase 'write limit' without incurring too many allocation locks. Increasing 'write_limit' simply places a higher loading on single sub-systems and turns this into a problem of reducing 'allocation_locks', which was covered above.

Cross-pollination Between Disk Operations and Scheduling

Disk operations are affected by the time slices awarded to active and eligible processes. These effects can be seen as excessive 'allocation_locks' and 'run locks' and occur for the following reasons:

1. When a process is awarded a time slice it executes and requires pages to be read or created whenever a reference out of the current set of pages in memory occurs. It does not matter that a page gets many references or a single reference, unless the program has not referenced the page for the current 'lap time' period. Thus a process that is awarded a very large time slice references many trivial pages that will not effectively take part in 'page attrition' until that process loses its time slice for longer than a lap time. Each of these pages must 'evict' an older page, and in turn all modified pages must be re-written back when 'evicted' after eligibility is lost. This tends to cause excessive paging and page writing, thus clogging the disk queues.

2. When a process has a long time slice, in a relatively short realtime period it will cause a large number of pages to be brought into, or created in memory. This tends to 'evict' older pages in a fairly compact group and to place 'new' pages relatively close together in the core map. When these pages must be 'evicted' later, if they are modified data, a very large group of them must be re-written together before a non-modified page can be found and 'evicted'. Since 'page_control_' cannot evict a modified page until the write is complete and the realtime process of writing is lengthy, this produces very large write bursts. If 'write_limit' is less than 58-63 these write bursts are seen purely as 'run locks', but if 'write_limit' is larger than 63, 'allocation locks' can also occur.

Setting smaller time slices gives each process a smaller time to attack other process's pages, and tends to produce a much more effective **'working set',** raising both system efficiency and increasing system responsiveness. [1] Though you may see higher IO rates, you should also see fewer 'locks', unless you have a 'thrasher' who can reference an incredibly large working set in extremely small cpu time, such as an inefficiently written editor or data manipulator.

Living With the Inevitable - Thrashing

If you cannot service the users of the system with small time slices without causing excessive MP idle and very high IO rates, then you are so deep into thrashing that the whole approach must be altered. In this approach, you actually increase time slices to approach 'batch' execution, and you simply give up the idea of responsiveness. At this point you are attempting to get the highest possible page utility out of each IO request, and to do this, you must grab as much of a single process into memory to be worked on as possible. This is an attempt to encompass the working set of the process in a situation where simultaneous 'attack' by other processes erodes pages too quickly. Allocating large time blocks permits a process to 'tickle' its pages continuously (hopefully continuously enough) to retain their utility and hold them.

Multics OPERATIONAL MODEL AND TUNING

Multics is an operating system which has been set up with a number of scheduling controls implemented to control time slices, guaranteed amounts of processor use, and separation of users into work classes. The manipulation of these controls, and the allocation of resources to interactive, absentee, and daemon or realtime users is generally the concern of the systems tuner and/or system administrator. Good manipulation can help overloaded systems to perform more efficiently, and very efficient systems to get better response and retain their efficiency.

This section in the tuning paper deals with the fundamental parts of the tunable Multics system, and presents a simple model for memory use that can be used to predict some parameters of system performance and point out bottlenecks.

Some information will be a repeat of information given in the 'Disk Tuning' section, and some will be background for it. The attempt here is to form a complete picture of the actions of tuning on performance.

Eligibility Queue

The eligibility queue is the queue from which processes can be executed. The limits of this queue are governed by 'maxe' and 'mine' and limit the number of simultaneously executable processes. Processes in the eligibility queue are picked to run from the head of the queue when a process currently executing becomes blocked for a page request.

WORKING SET OPTIMIZATION

Between the limits of 'mine' and 'maxe' processes being picked from the **Interactive** queue, **Percent Work Class** queues, or **Deadline Mode** queue must fit within the limits of currently assigned memory before they can become eligible. This memory limit is determined through the action of the scheduler (**pxss**) and the **'working set'** determined through the program **'post_purge'.** If the process picked to become eligible will not fit, then nothing is scheduled from these queues until either a better process comes along, or enough currently eligible processes lose eligibility to free sufficient memory.

'Working set' optimization does not optimize a system for small processes to produce better response etc. as much as might be desired. The only manner in which this optimization occurs is if within the 'best' work class another process gets sorted above the large process, or if another work class gets better credits, or if the interactive queue is in operation and smaller processes get to its head. If the interactive queue is the queue that has the large job at its head, then the entire system will wait, only running realtime processes, until sufficient memory is uncovered to run the large process.

Realtime processes do not have this working set restriction, they are made eligible immediately, if there is space in the eligibility queue, or at the end of their guaranteed response time even if 'maxe' would be violated. Realtime processes do not undergo a memory limit check.

Since they may be made eligible even if 'maxe' is violated, realtime processes are somewhat dangerous to overload a system. They can cause an extremely high instantaneous loading on the system.

Work Classes

Work classes are a division of the logical **Multics** workload into users grouped together due to common characteristics and/or to simplify administration and resource guarantees. Work classes, and their membership, are set up by the system administrator. Their membership, as well as their guarantees and processing characteristics, can vary from shift to shift.

REALTIME WORK CLASSES

Realtime work classes are those processes that will be granted machine resources as rapidly as possible, without regard to eligibility or core limitations. They are granted response guarantees and quantum guarantees, and are usually reserved for highly important processes such as the Initializer, daemons and possible realtime monitoring. They also tend to remain at the head of the eligibility queue during their period of eligibility to ensure superior response, while other processes are sorted into the bottom of the eligibility queue.

PERCENT MODE WORK CLASSES

Percent Mode work classes are the remaining processes in the system, when running in percent mode, rather than deadline mode. These processes reside in one of three queues:

1. The **Interactive queue** is a queue of those processes that have just finished an interaction and have become unblocked. The Interactive queue entries are always picked before any work class entries, and do not have any percentage guarantee limitations. It is a method to totally skirt around resource guarantees to attempt to retain system responsiveness.

2. The **Work Class Queues** are the queues assigned to processes in various work classes. Each queue has a certain number of time credits, in direct proportion to the percentage guarantee of the work class. Processes are picked from the work class queue that has the highest assigned credits, and are checked for memory size and 'maxe' violation. If the head process in the best work class queue will not fit then 'pxss' skips this scheduling period, and no process is scheduled. This causes the system to finish the eligibility quantums for enough of the other eligible jobs to fit the best process into memory.

3. Blocked processes constitute the final queue. These are processes waiting for some resource such as a tape mount, completion of terminal output, or waiting for terminal input. Processes going unblocked will either go to the work class queues, if they haven't interacted and are not realtime, to the interactive queue, or to the realtime queue.

DEADLINE MODE

Deadline mode uses the pointers for the Interactive queue to hold a list of all processes that are unblocked, not realtime and are waiting to be processed. These processes are sorted by the deadline guaranteed to them, and when made eligible, they will be given a quantum determined by their work class. Deadline processes undergo the same working set delay that percent processes have. If the process at the head of the queue will not fit, then no process is scheduled from the deadline queue and this scheduling round is skipped. This causes the system to stop making new processes eligible until enough memory becomes available for the process at the head of the deadline queue.

When Deadline Mode is enabled the normal work class queues do not exist. Response times and quantums are taken from work class information in the **'tc_data'** segment, as they would be for realtime processes.

The Credit Bank

In Percent Mode, credits are scattered from a variable termed the **'credit bank'** to those queues that have waiting processes, each time a new process is scheduled. The amount of credit to scatter to each work class queue is determined by the percent guarantee to that work class, and the credits in the bank. Credits are scattered in a ratio matching the percent guarantee until the bank runs out of credits. The credit bank is given credits to account for the amount of processing time used by all processes. This crediting process evens the distribution of excess credits over the demanding work classes according to their guarantees and tends to be fairly accurate on a loaded machine, where all work classes can fully utilize their guarantees.

Working Set Calculations

The use and calculation of a 'working set' value for processes is the combination of both 'pxss' and 'post_purge'. If 'post_purging' is enabled for a work class, and the system as a whole, then the page history for each process is used at the end of the eligibility to attempt to determine the amount of memory required by that process. This working set is highly dependent upon the paging history of the process and is easily skewed by a heavily loaded system, or a badly loaded disk drive. In fact just about anything, which slows or speeds a process out of proportion to other processes and the lap time, will skew the results of 'post_purge'. One can notice that interactive processes appear to have larger working sets than absentee processes and realtime processes since these non-interactive processes have not gone blocked and had page attrition affecting their pages. Since non-interactive processes may be able to come back quickly enough to access and hold their pages, they appear to have smaller working sets than is the actual case and are favoured by the scheduler.

Correspondingly, an absentee process, which is in a work class that does not have sufficient guarantees to hold its pages, will appear to have a working set that is larger than is the case and will be discriminated against, out of proportion to its true size. [2]

The working set value determined by 'post_purge' is used by 'pxss' in determining if the selected process from the Interactive, Deadline, or Work Class queue can indeed be scheduled, this is done by comparing its working set against the amount of remaining real memory available for paging. If the process will not fit, then scheduling is skipped for this round. Realtime processes are scheduled in two places in 'pxss' and do not undergo this memory check.

Lap Time - The 'Clock' Algorithm

Memory allocation is under the control of an algorithm termed 'the clock'. This nickname is due to the resemblance to the hands of a clock, where one hand moves to find a usable memory frame (the **'replacer'**), and the other hand moves to clean up pages that must be written back to disk (the **'purifier'**).

Usable pages are those which have not been modified, and have not been used since the last time that the cleanup hand passed them by, in other words since the last lap around the circular queue of the core map. Pages in this state are immediately 'evictable', since no work is necessary to ensure the disk copy is as up to date as the core page. Unusable pages are those that have been modified or used since the last lap of core management. This means that they must either be protected and the disk copy made consistent with the core copy before being re-usable (written from core), or they are in some other way unpagable (wired, out-of-service, etc.).

The time it takes the core algorithm to completely circle the core map is termed **'lap time'**, and is a measure of the length of time that an unused page will remain in memory before being evicted. Pages that have been modified will be added to the disk queue and considered to be evictable as soon as a completion is received for the disk IO. At that time core management places these pages at the Least Recently Used end of the core map (LRU) and they are evicted on the next step(s) of the core algorithm.

Paging Memory

Some statistics are available from standard system meters to enable the tuner to examine the actions of core management, primarily **'fsm'.** The 'fsm' meter indicates the lap time, the skipping of wired, used, and modified pages, and indicates the average number of steps needed to find an evictable page. These figures can give a rough idea of both how long a **think time** is available before enforced thrashing results, and about how much memory is basically 'locked' in place, either by references to the pages, or through 'wiring' of the pages.

On **The University of Calgary** system approximately 30% of memory is actually paged in the lap time, indicating that 70% is tied down by activity. This would indicate that as the system grows more active, less and less memory is available for the normal paging required to bring in non-reentrant code and data for programs. This is seen as a steadily decreasing lap time and steadily increasing response and disk IO. We consider that addition of more memory on **The University of Calgary** system would significantly add to this pool of 'Paging Memory' and significantly increase our lap time, and decrease our IO and response delays, **out of proportion** to the size of the additional memory.

Thrashing - And Causes

Thrashing is classically seen as the removal of a page of memory before its next use, requiring it to be paged back in again. The classical response to a thrashing situation is either constraint of the reason for the thrashing, such as a large program, or an increase in available memory, or both. We have found that thrashing can occur from a number of situations, some of which can only be solved by the above methods, but others which are subject to more subtle methods available to the tuner.

As can be seen from the section on **'lap time',** if a user can not access a page within a memory lap period, that page will probably disappear from memory, and will have to be paged in before the next reference. This can be considered a form of thrashing. Whether this is severe thrashing, or simply the normal attrition of sequential access memory depends upon the reason for the thrashing.

CLASSICAL LARGE PROCESS DRIVEN THRASHING

One way in which a process can thrash is that it attempts to reference a very large number of pages, which extends the eligible real time of all processes. This forces a large number of pages for other processes out of memory so that when the original large process loses eligibility and its pages start to trickle out of memory, a very large number of the other processes' pages have disappeared.

Realtime delay before the large process can come back into memory will be directly tied to the rate at which the disk system can process the page requests necessary to both write out modified pages and re-read old pages for now eligible processes. If the large process is delayed beyond the system lap time, it is by definition, thrashing totally. Lesser degrees of thrashing are also possible, but one must note that not only is the large process thrashing, the other processes in the system are also driven into thrashing. This would be typical of the action of the qedx and ted editors in a substitute or delete command with a large text file, or in other accesses such as a worst-case array reference.

THRASHING DUE TO LONG THINK TIMES

Individual interactive users can thrash themselves if they delay their interactions longer than the system lap time. In these cases the system feels somewhat loaded since modified pages must be written, but the thrashing will not be as severe as enforced driven thrashing, as seen above, since it does not thrash by driving out other processes' pages.

The only corrective action in the long term for this type of thrashing is to increase the amount of 'paging memory' to increase the lap time to an acceptable value.

Effects of Time Slices on Thrashing

Thrashing can be due to the effects of **trivial references** through excessive time slices, as was seen in the **"Disk Tuning"** section. This action occurs when a process is given a very large time to reference a large set of pages, with most references being trivial (only one or two accesses). In this case trivial references within a time slice will still force out some page and remove a possible future reference from core.

If too many trivial references occur before the owner of highly active pages can gain eligibility to reference those pages, and therefore hold them in core, thrashing will occur which is **driven by** the scheduling system. In this case all that needs to occur is that enough processes with large enough quantums delay the round-robining (**loop time**) of eligibility to longer than the lap time. The corrective action in this case is to reduce the length of time slices awarded by reduction of **'tefirst'** and **'telast'** so that the delay (number of processes in the cycle*inter-eligibility period) is less than the lap time.

Effects of Too Small Percent Guarantees

The same effect that occurs system wide, with time slices which are too large for the available memory, can occur if work classes are too heavily loaded and are awarded time slices which are too large, or percent guarantees which are too small. The effect of time slices that are too large was mentioned before and is seen in the round-robining within the work class. The effect of percentages which are too low is seen in the same manner since eligibilities will be granted less frequently, therefore driving the thrashing syndrome for the work class. In both cases corrective action is through decreasing time quantums granted to increase the frequency of the granting.

Further Words on Thrashing

As seen above, thrashing can be driven by the system through the awarding of time slices less frequently than the lap time of core management. This is seen as skewing the working set characteristics and obscuring the true LRU characteristics of the system. This can be corrected by the above-mentioned methods and by increasing the length of the shift register used to determine the LRU characteristics of the core map.

Another factor driving thrashing in the system is through the gradual degradation of time slice awarding as a process sinks within a work class due to the increasing **time since last interaction.** This tends to decrease the frequency of time slice awards and therefore drives thrashing by increasing the number of pages which will be evicted before the next eligibility. To be properly effective, this eligibility decrease should be countered by an increase in individual time quantums to permit a process to get more utility out of the pages within memory before they are thrashed out.

Effects of Small Quantums

At **The University of Calgary** we have been running a system with small time quantum awards for a number of months, both in periods of high activity and in idle periods. We have noted, in most cases, an increase in the amount of the system that can be delivered to the users, and in some cases a decrease in the disk IO. In almost all cases this has also reduced the amount of 'run locks' and 'allocation locks' which were occurring by better distribution of pages throughout the core map, and a decrease in the size of modified write bursts to disk. We have seen an increase in the traffic control overhead, from 3-4% to 5-8%, but this has usually been offset by the **5-10+%** increase in the **'Delivered to Other'.** On the whole the system seems to achieve a better working set and working set distribution.

When we are talking about small time slices we mean in the range of .07 to .15 seconds per eligibility. To this end we have had to modify 'ctp' to permit it to set 'tefirst' and 'telast' in values other than simple multiples of .125 seconds. (We have also modified 'pxss' to take percent mode quantums from the work class 'q' and 'iq', but this is not absolutely necessary.)

When we have experienced moderate to severe thrashing we have had to switch to large time quantums, as was dealt with in the **'Disk Tuning'** section, simply to achieve a workable working set for processes. Through very heavy periods we have experienced as much as 160 disk IO's per second for extended periods on an 11 MSU451 disk drive, 2 MPC (4 LA) system. (Our twelfth MSU451 is used for system source code and as a spare drive.)

Response Characteristics

Throughout our experience with small time quantums we have been able to experience acceptable response from the system, except in severe thrashing. By acceptable we would consider our system with 45-78 users and a 'tcm' response figure of .007 to .3 seconds as being acceptable. The users see this response as about 1-2 seconds on trivial commands, and also tend to see a steady delivery of system resources. This is in contrast to the good, fair, then lousy delivery that is more characteristic of too large time slices.

Small time slices also tend to move processes into the work class queues much quicker and prevent the syndrome where the system is loaded by the processes in the interactive queue and never gets around to process work class queue residents.

Large time quantums artificially delay work class residents' eligibilities and degrades their queue positions due to the increased time since last interaction.

Related System Settings and Modifications

We have also modified **'pxss'** to permit selection of **'wc_max_eligible'** while in percent mode, and have found this parameter to be quite useful in setting up the system. By limiting the possible 'over loading' by a single work class, we have been able to even out the instantaneous peaks in system loading, and to achieve much better distribution of resources.

This is seen in the prevention of too many processes attempting to get parallel memory resources from the system and competing too actively for pages. This tended to both thrash the memory system, and to be seen as a noticeable response delay until these processes could be cleared out. Limiting the number of eligible processes works well, with the percent guarantees, to ensure an even allocation of resources. This does not necessarily mean that we incur periods of too little eligibility to properly load and utilize the system, since this can be covered through both setting **'mine'** and appropriate **'twc'** settings in the **'shift_config_change'** exec_com.

An additional plus that controlled eligibilities has done for us is to enable the increase of 'maxe' to permit more effective multi-programming, and thus to drop the 'MP idle' which would otherwise run 5-20%. [3]

Our modifications to 'pxss', to permit initial and successive quantums to be taken from the work class information has turned out to be almost unnecessary since the use of very small time slices has tended to indicate that equal settings for quantums are the most effective and produce the best system characteristics. Thus we are effectively using a single 'tefirst/telast' setting of about 0.1 seconds.

AST Pool Tuning

The AST Pool is a collection of AST entries divided into four pool sizes (4, 16, 64, 256 **(255)** pages each), which is rather important in the operation of the paging system. The AST Pool holds one entry for each **'activated'** segment or directory, and there must be a complete trail from a leaf segment through all its parent directories to the root.

The number of pool entries in each pool governs the amount of 'mapable' memory that can be used in each pool, and also governs the amount of 'dead' memory removed from general use. The systems tuner attempts to make a balance between the size of each AST pool and system activity to provide sufficient AST resources while losing as little memory as possible in the necessary pool overhead.

One aspect of the AST pool is the amount of memory it can map, and another is in the characteristic demands of the users of the system. Typically one would have more AST entries than would be necessary to simply map a usable amount of memory, since a segment does not need to have any core allocated to be active, but does require an AST entry.

AST **'paging'** is seen when one does not have sufficient AST entries and the necessary activation of a segment requires a demand deactivation of another segment. Normally management routines will attempt to deactivate an AST entry that does not have any pages in core, but if too few AST entries exist this may be impossible. When this is the case, ALL the pages currently in core must be written to disk before the entry can be deactivated, this can be seen as demand writing which will tend to cause disk thrashing. An additional thrashing aspect of demand deactivation is seen in the deactivation of ALL links that have been snapped to the segment. This requires quite a bit of system overhead to track down all the references to the segment that will be removed from the AST, to snap the link back to a logical reference from a simple pointer. This is necessary to prevent a user attempting to use an invalid pointer or to prevent a reference that will be total nonsense. A final overhead created by demand activation and deactivation is in the **'directory management'** overhead necessary to re-activate the segment, and the VTOCE IO necessary to update the VTOC when deactivation is complete.

One typically looks for long enough lap times in the various AST pools to ensure that the overheads mentioned above are not critical in system operations.

A recommended minimum figure is about 200 seconds lap time, but large lap times, up to 1-2 thousand seconds are not severely large. Remember that this is again tuning for worst cases.

Parallel Loading - A Model of Memory Requirements

If one considers the operations listed above and the causes of system thrashing it can be seen that much of the operation, or lack of it in the **Multics** system is due to the effects of the load placed upon the Virtual Memory system. Much of this load is Independent of the number of processors available to the system, but is rather highly dependent upon the amount of memory and the number of processes attempting to use this memory in parallel.

As the number of using processes on a system increases, so does the time delay between successive eligibilities, **when this time increases above lap time the system is thrashing.** The extension of inter-eligibility time can come from one of two methods, either the processes cannot get enough processor resource, in which case one is limited to the number of processors available, or processes cannot get their pages fast enough and they are delayed by the real time read and write delays of the disk system. Since the difference in time ratios can be as great as 1:40000 or more, the effects of either too little disk IO capacity, or insufficient memory to hold an appropriate working set for the parallel process demand is quite adequate to outweigh the effects of too few processors.

Using this model for system tuning, one would attempt to estimate the memory demand of a typical mix of processes, and then determine the lap time and the IO for the system. Extrapolation to a desired point would arrive at a new number of processes. By taking the **'paging memory'** for the system, and a desired lap time one can arrive at approximately the amount of memory the system would be short of. This can be a very rough estimate of the amount of memory that this typical process mix would require.

Correspondingly, one can take the current mix, determine how much the lap time should be extended, and increase the system memory by the increase in lap time, multiplied by the number of pages turned in that lap time ('paging memory'). Again this is a rough estimate of needs, but it is usable.

When a rough approximation of necessary memory has been arrived at, one would attack a deficit in processor throughput by simply adding processors sufficiently to ensure enough processor cycles.

We have found that our two processor system, to gain a lap time of about 6+ seconds would require at least another million words of memory.

ROUGH ESTIMATES OF PERFORMANCE

Our past experience has shown that one could expect about the following performance levels, at the following lap times:

Lap Time	Delivered
3 sec	35-45%
3.3 sec	45-60%
3.6 sec	50-65%
4 sec	60-70%
5 sec	70-80%
9 sec	80%+

Conclusions

We conclude from the above information and past performance that the performance of a **Multics** system depends rather highly upon the efficiency and capacity of the **Virtual Memory** system. If there is insufficient memory, or it is managed in a way that obscures the true working set of the processes, one will be unable to extract the full amount of use from the system.

Our experience, with the **'p_ast'** tool indicates that rarely does one have more than about 16% of the mappable amount of memory actually present in core. This strongly indicates that **Multics** working sets are significantly smaller than the full address space and points out the ease with which a thrashing program can skew system operation.

The methods outlined above seem to have worked well at **The University of Calgary**, and we will probably continue to use them in the future. We are expecting to receive a significant increase in the capacity of our system in the near future (1 proc, 1.5 million words, 1 IOM, 1 MPC) and are interested in determining the validity of this model, admittedly developed with little cross-reference to other conditions than our own.

TOOLS AND THEIR USE

This section deals with metering and metering tools. It presents some of the more interesting tools that are supplied with the system, and also some that we have developed at **The University of Calgary**.

Tools for Monitoring Disks

There are a number of tools available for monitoring the action of the disk drives and determining bottlenecks. The list below is not necessarily in any order:

dq This is a standard system tool which lists the number of connects to each channel for each disk sub-system, and lists the core and disk references for each IO request in the sub-system disk queue.

find This is a tool developed locally, which provides **'dq'** like output, listing the actual files involved, rather than the memory frames, listing the number of pages in the queue, the pathname, and where possible, the **Person_id.Project_id.mode** and **tty_name** of the user using the file. This tool lists the channel connects, but also lists the channel number and channel status, for example: **Br** = broken, **Io** = assigned to IOI, and **In** = inoperative, but not yet broken.

Find is very useful in determining disk bottlenecks in terms of file activity and usually is able to pin this to a particular user, either by the files in use under the **Person_id.Project_id,** or directly to the terminal upon which the user is running, or the absentee id.

We have already used find to determine that qedx was giving us noticeable problems and to correct this situation. This is the subject of another paper given at this conference.

Find lists the file length, the number of nonzero pages, and the number of pages currently in core. Problem files will probably have an inordinately large number of pages in core and may also have a large number of pages in the queue. Anything over a single page in the queue generally means that most of the file activity is due to page modifications. It is generally harder to cause disk system thrashing through the action of demand reads.

dvm This meter is useful in listing sub-system activity in terms of channel use and IO's. It **is a** good way to get a comparative measure of read and write activity and balance between systems, and the amount of spare capacity that might be available. Other useful values are in delay times for reads and writes and VTOC IO, and in error statistics.

disk_meters This meter is useful in listing drive and queue activity. Drive activity is seen through the balance of reads and writes and the seek length of the IO's. What is also of interest is the balance between drives in a sub-system, which can be seen by comparing ATB IO, and the number of reads and writes.

Queue activity is seen through the header information for the sub-system, where the number of requests, the 'run locks', 'interrupt locks', and 'allocation locks' can be seen.

If one is suffering significant lock problems then comparing drive balances can determine the drive on which the problem is occurring, but **find** would be the best method to determine the actual file that is the problem.

We have noted that quite a bit of our loading problems can be traced to the IO skewing caused by the Volume dumper.

mwcm This is a locally written work_class_meters program that collects a large amount of information from other meters and presents it together. Drive information that would be of interest in this meter is seen as the number of page IO's, the current page IO rate, and the length of the average demand page read delay.

An additional value listed is the amount of memory that is currently taking part in paging during the lap time, but this is really a combination of scheduling and tuning.

intm This tool indicates the interrupt overhead distribution across the system, and can be used to examine the channel loading to the disk system. Here one would be expecting value of less than about 1.25% for the heaviest hit channel.

Virtual Memory System Meters

The performance of the Virtual Memory system is rather critical to the operation of Multics as a whole. The following meters are useful in monitoring that performance:

fsm **(standard system version)** This tool lists basic file_system information. Values of interest are divided into three basic areas:

The last section of fsm deals with core map and paging activity. Here one would be looking at the distribution of paging across directories and the system, and other interesting values would be seen in the lap time, the average number of steps and the distribution of step reasons.

Other values for core management are listed for claim runs and the number of pages usable and wired.

fsm **(locally modified version)** This is a locally modified version of the standard fsm (above) that outputs some additional values. It corrects the lap time calculations of the system fsm, and supplies a number of additional meter values in the core management area. Here more values are given for the action of the **'purifier'** hands of the **'clock'** management algorithm, including the number of pages written and the skipping of the **'purifier'.** Other values of interest are the distribution of wired pages and the current **'write_limit'** and outstanding write count.

The modified fsm can supply values for the amount of 'write_thrashing' that the system is undergoing through the 'Mod during Write' value. This is an indication of the number of times a page was queued to be written but was accessed before the write was complete. Since the write had to complete and tied up disk resources this is a measure of the amount of overhead the system is undergoing due to thrashing of pages being written.

An additional value is the 'New pages' count, which indicates the number of pages which had to be created, and the number of pages which were 'Zero for write', indicating that an access was made, but probably was simply an initialization to zero.

mwcm Again this tool enters into things by listing AST pool information, as well as lap time and page faults.

p_ast This locally written tool is a counterpart of **find,** and is useful to examine the AST pool. It has a number of control arguments to control the type and extent of AST examination. One of the more interesting outputs is the pool statistics, which lists the utilization of the pool entries, the amount of memory used for directories and segments, as both number of pages, and percent of available memory. It also lists the amount of memory that the AST entries could map, and the amount of memory actually in core.

These values can be rather useful in determining what the AST is being used for, you will find a lot of directories, and whether one should decrease the number of entries without suffering problems with deactivations and deactivation page thrashing.

Other outputs from p_ast will list the files addressed by the AST, giving the same output as find, with the exception that the first two values are: the AST entry number within the pool, and the disk drive the file is on, rather than the disk drive the file is on and the number of queued pages to that drive. Here the AST entries printed can be selected from those segments and directories with core assigned, directories, or segments, or both.

By examining AST pools for occupancy of core, one can see things that might not be as easily seen through find, unless disk activity was occurring. For example most thrashing would occur from files in the 256-page pool. Thus examining all those files which have core assigned would enable one to see those files which may have too large a number of pages in core, which could lead to contacting a user to correct a problem BEFORE it becomes an operational problem. You may also find it rather interesting simply to see what is there, particularly during different loading conditions.

Scheduling and Performance Meters

The performance of the scheduling system can be seen through the following meters:

ttm This meter gives one an idea of the overheads and the delivery of the system as a whole. It is useful to pin down bottlenecks and possible problems. By breaking up the sources of idle time it also lets one see if the idle being experienced is an artifact of too light a loading or too heavy. One of the most important figures here is 'Delivered to Other', which one would like to make as high as possible, but other overheads may need to be kept low. For example: on our system if 'Paging' starts to exceed about 8-10% we know that we will start to get bad (20-90 second) initializer response, and that values over this will definitely give us bad login/logout response, and will indicate a bottleneck either in the Virtual Memory system (thrashing) or in the disk system (drive overloading and allocation/run locks).

tcm This tool provides information about the traffic controller and operation of the scheduler. One of the interesting values is the interactive response time, which is a measure of the time delay from becoming unblocked (and going into a workclass or interactive queue), and actually becoming eligible. Another value of interest is the average number of processes in the eligibility queue, which should almost always be below **'maxe'.** When this value is at or above **'maxe'** this is a strong indication of too many realtime processes, or an exceedingly delayed system. The second situation can be attacked by decreasing the time quantums awarded to shorten the eligibility period.

tcq This is one of the more classic traffic controller meters. It is useful in seeing the distribution of system resources to users, particularly paging to cpu ratios and the calculated working set factors. Another purpose for this meter is in the displaying of the average loading for the work class queues, and through the **'time since last interaction'** value a means to measure if thrashing is taking place within a work class. Additionally it prints the credits assigned to a work class, and so a comparison of the work class loading in relation to the available credits starts one on the way to seeing if thrashing is taking place due to insufficient guarantees.

At **The University of Calgary** we have run without the eligibility queue enabled for a number of months, in running with small time slices, and find this to be a superior means to achieve good distribution without loss of good interactive response. Here we maintain the good response by limiting overloading through excessive eligibilities and by small time slices.

Final Words on Metering

Metering and tuning is almost an art form in that it is rather personal in the ways in which it is carried out. Different people will see different things in different meters, even with identical numbers, who is to say that they are incorrect if they can produce a tuned system.

Much of the above information is my personal view and hinges upon the methods used at **The University of Calgary**. I hope that these methods and views of performance will be able to expand your own views and improve your own situations.