This document is meant to give a description of the proposed alteration to the Multics virtual memory implementation often referred to as either "drum multi-level" or as will be used in this write-up, "page multi-level". The reasons for undertaking the proposed change are many. The prime reason is to take fuller advantage of the "small" latency time and fast transfer rate of the current drum. Under the current organization the drum rarely is taxed to 25% capacity. This problem is not only a serious waste of resources but will likely be amplified by an even faster device as is being proposed for the follow-on hardware. Whatever device is available, be it drum or bulk store of some type, there will be precious little of it and it will be up to the system software to take full advantage of it. Leaving a rarely used page on the fast device is clearly not the most optimum approach. The rest of this write-up describes a general approach to take which has several options at a detailed level. Before describing the detailed plan, however, it should be stated that current meters in the system clearly show that the main system bottleneck is disk channel capacity and disk latency time. These same meters will obviously be chosen to test the success of whatever page multi-level scheme is implemented. However, there should also be several other meters in the system better designed to measure the actual changes that should result from the page multi-level scheme. These should include some measure of "drum residency time", some measure of the success of the replacement and displacement algorithms, and some measure of any overhead brought about as a direct result of the page multi-level implementation.

The Basic Philosophy

The page multi-level scheme is a departure from the current Multics virtual memory implementation. Currently, a page is either in core or it is not. If it is in core there may be a valid copy of the page on "secondary" storage. Secondary storage currently is everything other than core memory and includes the drum and the disks and could logically

include tape storage for "removed" files. The page multi-level scheme would introduce a third logical storage device between core and secondary storage. This device (the drum in our current plans) is meant to hold a copy of the n most recently referenced pages in the system where n is the number of records on the drum (or whatever device is chosen). (The actual algorithm for deciding which pages will be placed on this device will be discussed later.) In an analogous fashion core memory contains the M most recently references pages where M is the number of pages of core available. In other words core pages are stand-ins for drum pages which are in turn stand-ins for pages somewhere in secondary storage.

We will speak of the drum (or whatever device is used to hold the recently referenced pages) as the "paging device". The basic goal of page multi-level can be stated simply as keeping the most frequently referenced pages on the paging device.

There are two basic questions which the system must dynamically answer in order to implement page multi-level. The first is, "When do we place a page on the paging device?" and the second is, "Which page must be moved off the paging device when a record of the paging device is needed?" These questions must be asked while processing a page fault and hence the information needed to answer these questions must be wired-down. It is, of course, best to minimize this wired-down data but it is also important to keep enough information around so the scheme not only performs well (keeps most paging from the paging device), but also performs efficiently.

The algorithm used to decide when a page is brought up to the paging device should be simple and easy to implement. At first the algorithm will probably be to bring a page up whenever it is referenced and is not on the paging device. However, there are several refinements which may be worth thinking about and will be listed below.

1) Don't bring a page up until its second reference within a certain time interval.

2) Have a primitive (file system) which can control how the pages of the segment are treated.
   a) Treat as sequential file.
   b) Don't move up at all.
   c) Move up and keep up, etc.
3) Don't bring up pages of a "sequential" file. Let the algorithm try to determine if the page references are sequential -- from zero, etc.
4) Only bring a page up if it is modified.
5) Only bring a page up if it is in the process directory or shared, etc.
6) "Bring up" every newly created page.

Some of the above recommendations are probably unreasonable not only because they are too expensive, but also because they go against the basic strategy by trying to second guess the system rather than let the system tell you what it's doing. These are listed mainly to illustrate the possibilities.

The algorithm used to decide which page is to be moved down when another record of the paging device is needed is basically a least-recently-used algorithm. That is, the page which has gone the longest without a page-fault (and which is not in core) is chosen to be moved down. In a manner similar to that of the core replacement algorithm, information will be kept as to whether the page has been modified since it was brought up and if not, it is probably not necessary to move the page from the paging device to secondary storage.

There are two ways of keeping track of which pages have been used recently and which have not. The first is to keep a threaded list of all pages on the paging device with the head of the list being the least most recently used page. The list will be kept up-to-date by moving a page to the end of the list whenever a fault occurs on it and by placing

free pages at the head of the list. A second approach is to approximate the least-recently-used algorithm by using a counter for each record of the paging device. This counter could reflect the activity of a page in several ways the details of which need not be spelled out. The table of these usage counters would be scanned sequentially (by record number) when searching for a record to be used.

The main difference between these two approaches is the amount of storage they would require. The thread approach would require a two-way thread. The counter would likely be several bits wide.

An important decision to be made is whether or not to implement the removal algorithm in wired-down code or not. When a record of the paging device is needed (at page-fault time) there are two approaches which could be taken. First, page control could look for a free record (or at least one that need not be updated onto secondary storage) and then use it. In this search it may be necessary to initiate the mechanism to move several pages down by first reading them into core and when that read is complete writing the page out onto secondary storage. This read/write sequence will be necessary no matter what, the question is whether it should be initiated by page control or by some other (non-wired) program. The second approach is to have this read/write sequence initiated by some other program at specified times. This approach would work by keeping a free list of paging device records which is replenished when it is noticed that the free list is too small. The free list itself must be wired-down but the code to replenish it need not be. This second approach has the advantage that it avoids an interesting recursion problem in page control. When page control initiates the move down it requires a block of core to use. However, page control is right in the middle of the code to find a block of core when it runs into the move-down requirement. If another program were to request the core and initiate the read/write sequence -- independently of page control -- the recursion is avoided. The disadvantages to the second approach are first, part of the paging device will

always be free (not in use) and second, there will be times when page control would like to move a page to the paging device but there are no free pages. Page control would have no choice but to leave the page on secondary storage until some paging device records free up and a page fault occurs on the page in question.

The programs which would keep the free list full could possibly be the AST management programs. They could check, say, whenever a segment is activated to see if there are enough entries in the free list. If not they would initiate the appropriate read/write requests to again fill up the free list.

The algorithm to move a page down can be arbitrarily complex. It could use information analogous to what was mentioned above for the move-up criterion (in addition to the activity and/or use of the page). It would be easier to implement a more complicated algorithm (if so desired) if the algorithm existed in paged (PL/I) segment control code.

The read/write sequence alluded to above requires changes to page control at "posting" time. The core map entry associated with the block of core being used to perform the move-down must contain a flag which differentiates the current use of that block of core from normal use. This flag effectively says, "Instead of turning access ON when the read completes, initiate a write to the specified secondary storage address." When the write completes the flag directs page control not only to thread the core block into the front (free end) of the core map but also to mark the paging device record as free (by whatever technique will be used -- i.e., free list or free end of usage list). Note that both device addresses are needed at posting time.

The Device Address Format

The device address currently consists of a record number and a "moved" bit which says whether or not this page has yet been "moved" (in the file multi-level sense). The device ID is currently extracted from the AST entry (as is the move device ID if the moved bit is ON).

With page multi-level there will be times when we have a device address but do not have an AST entry for the segment to which the page belongs. For this reason the device address must be extended to include the device ID as well. The need for the moved bit goes away as we no longer need to decide which device ID in the AST entry is appropriate. If a segment is being moved (in the file multi-level sense) the move device ID is set appropriately in the AST entry and each page is brought into core. As page control writes the pages out of core it deposits (frees up) the old addresses and withdraws new addresses from the free list of the move device. Page control places the move device ID in the device addresses it creates for the moved pages. For page control to determine whether a page has been moved or not it must merely compare the device ID in the device address with the move device ID in the AST entry. At deactivation time it will be noted whether or not all pages have been moved and if so it will mark the move as complete by copying the move device ID into the device ID and zeroing the move device ID.

The device address will, therefore, include both a record number and a device ID.

Changes to Directory Control

The new device address format means that the file maps in directories must be changed. Instead of the current 18 bits a device address will now contain 36 bits. Not all of these will be used by page control and it is recommended that the unused bits which will exist in the file map be used to carry redundant information which can be used to avoid re-used addresses when hardware errors occur. The fact that directories must change means that we are at a convenient place for doing several other directory reformatting jobs. In particular, retro-fit part (3) (allocating file maps as a function of size) should be done at this time. In addition, it is recommended that the following be done at the same time:

1) Do whatever is to be done with respect to CACL's.

2) Change the charging scheme for disk records used.

3) Leave handles (only) for

   a) new backup requirements

   b) new multi-level requirements

   c) tape archiving

It is recommended that no changes be made for the explicit use of secondary storage reconfiguration. The main requirement here is for all file maps in the system to be localized. This is too large a change for now as there is no immediate need for secondary storage reconfiguration.

Deleting Segments

A problem arises when it is desired to delete a segment which has pages on the paging device yet which is not active. Since the file map in the directory will contain the secondary storage device address there is no obvious way to find the paging device addresses for any pages which have been moved up. The proposed solution is as follows: first, the file map in the directory will contain a flag for each device address which specifies whether or not the page exists on the paging device. If so, the secondary storage device addressed is hashed to come up with an index into a hash table. The hash table entry will point to a list of entries in the paging device map which have the same hash code. This list will be searched for the appropriate entry (which contains the secondary storage device address) which will then be "freed".

It was mentioned above that the secondary storage address would always be in the directories. When a segment is activated this address is placed in the page table word until the page is brought into core in which case it is placed in the core map. If the page is written out onto the paging device the secondary storage address will be copied into the paging device map entry and the paging device address will be placed in the page table word. From this time on any page faults which occur will read the page in from the specified paging device record. If the secondary storage device address is ever needed (for a given page) it can be found by going to the appropriate paging device map entry (indexing by the address found in the page table or core map entry).

When a segment is deactivated it must be determined which of its pages exist on the secondary storage device and set the flags in the file map appropriately.

Data Base Organization

The data layout needed to implement the page multi-level scheme requires changes to the following data bases:

1) core map — must contain extended device address.

2) directories — must have new file map format. File maps should be allocated as a function of size. (Other unrelated changes which are convenient.)

3) pd map — (paging device map) a new data base which must be wired down. It will contain either:

two-way thread	24-bits
hash-thread	12-bits
device address	24-bits
modified bit	1-bit

or

usage bits	n-bits
hash-thread	12-bits
device address	24-bits
modified bit	1-bit

This must be wired-down.

4) pd hash table — paging device hash table. This must be wired-down.

The page table word must be organized so that a 24-bit device address will fit into it if the page is not in core. This is reasonable on the current and the follow-on hardware.

Pre-paging and Post-purging

The only change which affects pre-paging and post-purging in page multi-level is the way the "drum decision" is calculated. One of the criteria for pre-paging a page is that it be on the drum. This information must now be extracted from the device address itself rather than going to the AST entry.

Emergency Shutdown and Salvaging

With a good deal of the Multics hierarchy residing on the paging device it will be imperative that emergency shutdown and the salvager work almost every time. Since it is not too unlikely that core memory is lost the information needed to recover from a crash should frequently be updated onto some more reliable device. In addition this updating should not tie up any valuable resource. Since the data to be updated is in the neighborhood of 10 pages and since we would like the update done no less frequently than once every second or two the logical device to choose (currently) is the drum. In fact, the drum can easily be programmed once per bootload to automatically update these pages every second or so.

When emergency shutdown runs it should do as much of the following as it can:

1) Write the pd map onto the drum.
2) Update the pd map in core with the most up-to-date information from the AST entries.
3) Write the pd map onto the drum again.

As a final step in the shutdown process (after all paging has been completely and all data bases are consistent) the pd map should be written out one last time.

The salvager must likewise recover as much from the pd map as it can. The salvager will know whether or not an emergency shutdown was run and how

successful it was (whether it completed). At any rate the pd map will be no more out-of-date than a second or two. The salvager must do the following in addition to its current tasks:

1) Valid and recreate, if necessary, the pd hash table.
2) Valid and recreate, if necessary, the pd free list (if being used).
3) Make sure that the file maps in each directory accurately reflect the fact that some pages have a copy on the paging device.
4) Update onto secondary storage all pages which are marked as modified in the paging device map.

Note that the salvager must be able to find the (dedicated) records of the paging device map. This information must be kept in the fsdct.

Programs to be Affected

The following areas will be affected by the page multi-level redesign:

1)	page control	page, pc
2)	segment control	activate, deactivate
3)	directory control	extensive reformatting
4)	initialization	tqut, uct as branches pd map on pd.
5)	emergency shutdown	as noted above
6)	salvager	as noted above