To: Distribution

From: Bernard Greenberg

Date: July 5, 1976

Subject: Nulled Addresses, Deciduous Segments, Etc.
A Primer in the Page-and-Segment Newspeak of NSS

The implementation of the New Storage System (Release 4.0) has added to the already devastating plethora of arcana known as Multics by creating many new concepts and new terms for old ones. In particular, the handling of disk addresses and initialization segments has created quite a bit of confusion among the uninitiated, and deserves some explicit clarification. This memo attempts to alleviate some of this confusion, and enlighten the interested. The explanations below are applicable to systems of MIT genre 28-5a and later.

In the old storage system, all disk space was part of a common pool, shared by all segments of all natures, access classes, and places in the hierarchy. What is more, all disk storage was always present at once. When all of the disk storage in the system was used up, that was it, there was no more, and the system had reached a fatal error condition. In the new storage system, each logical volume (user visible set of physical volumes (disks)) constitutes its own pool of storage. If a user logical volume runs out of space, this is the user's problem. It is an error similar to record quota overflow, causes by mismanagement of resources. A user encountering such an error should be able to delete other segments on that volume, and continue. It is no occasion for a system crash.

In the old storage system, disk addresses were allocated at the time a page was written out from core. Many pages simply come into core and are zeroed or truncated, such as mailboxes, and never have to be written out. Thus, postponing allocation until write time was a good policy, as it reduced the withdraw-and-deposit (1) traffic, which in the old storage system might contribute to reused addresses should a crash occur, and

---

(1) Withdrawing a page is asking for an allocation of a page, and having it allocated from the appropriate free pool. Depositing a page is returning it to that free pool.

avoided doing potentially useless work. One very broad design goal of Multics in general has been to delay bindings until the last possible minute.

In the new storage system, we cannot afford such a luxury. The old storage system was well aware that it needed not allocate until write time, for the only case in which that allocation might fail would be a fatal system problem from which only the most inelegant recovery was possible.

Since we cannot be assured of the availability of a page at write time, we cannot create a page unless we can commit the resources necessary (a disk record) to it at page creation time. This way, no data has been lost if an allocation cannot be performed, just as with a record quota overflow. (By the way, Record Quota is not a technique for making sure the system doesn't run out of disk, but rather a tool to enforce administrative policies on the sharing of the disk resource: that is, to make sure no one takes more than his allotted share.)

Therefore, the new storage system assigns addresses to pages at page creation time. Since page creation only happens during page faults, the faulting process can be notified if page allocation fails, via the mechanism of a signal emanating from the page fault (as is done with quota). Now when an address is so assigned to a page, the page has, upon the completion of the page fault, both this disk record and the page of core associated with it. Similarly, a page which has just been read in from disk has a page of core and a disk record associated with it. Yet, the first case is different from the second, as in the first case, the data on the disk record does not correspond to the page of the segment, while in the second case it does. Assuming that the faulting reference did not modify the page, the first case has zeros as the contents of the page, while the second has the contents of the disk record. In page control, we mark a disk address of the first type with a special bit:

> A nulled address or semikilled address is an address of a disk record, which is assigned to a page of a given segment. The contents of the page of the segment, however, are not the contents of the disk record, but zeroes.

The above definition of a nulled address shows that it is akin to the concept of a device being assigned to a process but not attached. No one else can use it, it can be used by this page/process whenever it needs it, but it is not now using it.

In the old storage system, a major cause of misrouted data and resultant grief was the so-called re-used address phenomenon, which consisted of two segments claming the same page. This would often happen after a crash, when someone who had zeroed or truncated a page would mark at as free (in the

appropriate volume's free pool), and it would be allocated by someone else before it was take out of the file map for the previous segment. In the new storage system, this is impossible, as addresses are never returned to the free pool (deposited) for their volume until the VTOC Entry (VTOCE), which contains the Segment File Map has been appropriately written back to disk. This precludes all address depositing by page control at write time. Thus, when a page is discovered to contain zeroes by the core replacement algorithm, or truncated, we would somehow like to associate the disk address with the page, but zeroes with its contents. Sound familiar? That is precisely the definition of a nulled address! Hence, zeroed and truncated pages revert to the nulled state.

When the VTOCE file map is updated for any reason (deactivation and the AST housekeep known as the "AST Trickle" are two), any addresses culled from page control which are seen to be nulled are not reported to the File Map on the volume. From the definition, your segment has no right to the contents of the disk record. Furthermore, if your page is not in core or on the paging device, it is a certainty that its logical contents are zero. Thus, we allow you the privilege of recreating the page by depositing the address at this time, replacing it in the Page Table with a null address (defined below, not to be confused with a nulled address). How does one acquire a right to a page? Right to a page by write to a page! When page control knows for a fact (successful completion of a disk write) that the page of your segment, to which a nulled address was assigned was written to the disk, the following happens:

> The resurrection of a nulled address is the removal of the nulled flag, logically associating the contents of a disk record with a page of a segment. Resurrection happens at page write complete time and Read-write-sequence (bulk store flush cycle) completion time, and establishes a segment's right to a page of disk. A resurrected address is known as a live device address. (1)

Now when segments are created, no disk addresses are associated with them. After a segment is truncated, and appropriate deposits have been performed, again no addresses are associated with the pages. Thus, there is a large conceptual device all full of zeroes on which all pages have addresses assigned (non-uniquely) when created and destroyed. Such "addresses" represent no disk storage, assigned, attached, or

---

(1) A feature in system 28.7 and later endows the Physical Volume Salvager with the ability to resurrect addresses found nulled on the Paging Device (Bulk Store), if the paging device was not flushed by the previous shutdown (as in a crash in which Emergency Shutdown failed).

otherwise:

> A null address is an "address" which is not the address of any disk record, but rather a designation of a source of infinite zeroes. A page which has a null address assigned is not associated with any disk record in any way, but with all zeroes.

Null addresses in fact contain identifying codes as to their program of origin, so that the maintainer of page control can isolate and fix many different kinds of problems.

VTOC entries (file maps in particular) are not into any of this semikilled-nulled mickeymouse. A VTOC Entry says, "Either this page is yours, or it's not." Hence, only null and live addresses appear in VTOC entries. As we said before, the VTOC entry updater puts a null address in a VTOC entry ("You don't own that page!") when he sees a nulled address in the page control data bases. When a crash occurs wherein all of main memory and/or bulk store is lost, the VTOC will in fact say that no one owns any page to which he has not in fact written.

- - - - - - - - - - - -

In the new storage system, each segment is constrained to live on one pack (physical volume). This was a major design point of this version of the storage system, as a frank view admits the pack as the unit of storage system failure. Therefore, free storage pools are maintained per mounted physical volume. Therefore, when an address must be assigned to a page of a segment (remember, all addresses are withdrawn as nulled), the Active Segment Table Entry of the segment is interrogated to identify the drive on which the correct physical volume is mounted, and the free store map (1) for that drive withdrawn against. If no more records exist for that drive, the segment cannot grow and an error is signalled.

BUT WAIT A MINUTE! Aren't physical volumes supposed to be transparent to the user? Aren't logical volumes the user-visible entity and storage pool? Well, given the constraint that each segment must reside on a physical volume, that puts us in a tight dilemma. Clearly if there is any physical volume in the logical volume which has enough space to hold the segment, it ought be moved there. If there is none, the user ought be told:

> The segment mover is the program which tries to move a segment between physical volumes of a logical volume in

---

(1) The free store map (or volume map) of a pack (physical volume) is the bit map of the Multics pages on the pack that says which records are in use and which are available for allocation. Addresses are said to be withdrawn and deposited against the volume map.

the case of physical volume overflow (also known as oopv, for out of physical volume). If no physical volume in the logical volume can hold the segment, logical volume full is signalled (subcondition of seg_fault error).

The segment mover is an extremely baroque and complex program, which calls dozens of entries in page, segment, and VTOC control and plays a myriad of locking and cleanup games. His name, oddly enough, is segment_mover.

- -

Initialization has been an anathema for many Multicians, due to many clever bootstrap techniques, tricks, and special-case policies of interest only to the specialist. The management of segments by initialization has always been in need of more illumination to the multing public (although the Initialization PLM, AN70 has helped many to understand), and the changes of the new storage system to these policies have been deep enough that a new explanation is warranted.

The Multics hardcore supervisor is read in from a tape at bootload time. Since it is too big to fit in main memory at once, and Multics is a virtual memory operating system, it is divided into two collections (purists please note that Collection III is not the hardcore supervisor), the first being all that is necessary to establish paging, and the second being all else. Paging is a teriffically complicated business that involves all sorts of knowledge about what disks are where, data bases, maps, pointers, chains, and algorithms, not to talk about the tacit support of all of the ring zero I/O management mechanism. All this preassumes a completely pre-linked PL/I world at that. Thus, during the loading of Collection I, all of the software necessary to initialize and carry out these functions must be put into unpaged segments, because the software to establish and maintain paging hasn't been established or maintained yet. The hardware supports segments that are contiguous in main memory via a special bit in the segment descriptor word (SDW). Unpaged segments are not to be confused with wired segments.

An unpaged segment is a segment which occupies a contiguous region of main memory, corresponding to all of the addresses of the segment. Only pieces of the supervisor can be like this.

Many pieces of the supervisor, specifically those that handle interrupts, paging, and traffic control (scheduling/multiprogramming) cannot themselves be subject to paging, due to eventual recursion and finiteness problems. Such procedures and data bases must be assured to always be in main memory. For unpaged segments, this is always the case. Multics never removes unpaged segments from memory (although see below).

For certain paged segments, the paging control software is restrained from removing their pages from main memory by special calls resulting in the setting of special bits.

A wired segment is one that cannot be removed from main memory. Either unpaged segments or paged segments that contain wired pages (pages that cannot be removed, by covenant with page control) can be called "wired".

A paged segment is one that has pages, i.e., is not unpaged. The segment descriptor points at a page table, and contains a bit to this effect. The contents of the segment are located by the appending unit through the page table. It may be wired or pageable.

A pageable segment is a paged segment which is not wired, i.e., whose pages can be removed from main memory at the discretion of the page control algorithm. All user segments, and all directories are pageable.

We have stated that all of the segments read in in Collection I are unpaged. Although this is true, there are a certain subset of these segments which are unpaged only because that is the only way one can be in Collection I loading. For instance, the program that searches configuration decks is needed in Collection I to ascertain configuration information, but need not be wired, much less unpaged. Another example is the Segment Loading Table and its Name Table. Therefore, at the completion of Collection I, an event known as the making paged of the segments occurs, conducted by a program called, appropriately enough, make_segs_paged. All of the segments which are to be made paged were placed by the Collection I loader (bootstrap1) at the high end of main memory, while those which are to remain unpaged were placed at the low end. (A couple of peculiar stragglers, known as "Collection 0", created by bootstrap1, appear at the low end, but are made paged).

The hardcore supervisor is much like a process unto itself. The SDW's which describe its segments share the same segment number in everyone's process, and appear in everyone's process. Only a handful of segment numbers (those associated with the names dseg, kst_seg, pds, and prds) denote different segments in different processes. Once any process gets into page control and locks the lock, it is as good as any other. This behavior has been declared varyingly a bug and a feature by various students of system architecture.

On certain instances, such as dealing with directories, the hardcore supervisor actually comes up with pathnames, and initiates segments, just like your programs. but most of the

time, it deals with segments that were never initiated, linked, or activated. The hardcore supervisor is a world of segments loaded off of a tape and created at bootload time, prelinked at that time. The paged segments cannot be deactivated, nor SDW's faulted, nor can wired segments be removed from main memory. All of the necessary SDW's appear in everyone's process, and nothing need ever be initiated.

There are certain segments that the hardcore supervisor uses that other people might like to use too. For instance, as the supervisor is coded mainly in PL/I, and uses stacks, the supervisor has a segment called "pl1_operators_" that it uses for these language support functions. If it's so useful, why can't everybody share it? Also, there are the hardcore gates, whose things you have to snap links to for your process to crawl into the hardcore supervisor. There are also a couple of data segments wherein the supervisor provides you with certain information about himself, such as sys_info and active_all_rings_data. The only way anybody except the supervisor can ever find these segments is for them to be placed in the Multics Storage System Hierarchy, by having a branch, a VTOC entry, and a unique ID.

> A <u>deciduous</u> segment is a segment which comes in off of the system tape (and is thus part of the hardcore supervisor), and is also in the hierarchy.

Deciduous segments are so called because, like deciduous plants, they periodically lose their leaves (pages). Every bootload creates the deciduous segments anew as they are read in, voiding the previous ones. Deciduous segments are not "activated": they are active before a concept of activation exists in a given bootload. The program init_branches creates empty segments at the place in the hierarchy where deciduous segments are to appear, but instead of activating them, <u>connects</u> the branches so made to the supervisor segments which are to become deciduous via the AST UID hash mechanism, by which potential activators of segments learn of their activity. The net effect is that anyone who takes a segment fault on such a segment, having initiated it, finds it active, its page table and ASTE being those created of the world of the hardcore supervisor by initialization. The hardcore/user segment sharing mechanism works thus.

- -

Another form of hardcore/user segment sharing is seen in the management of Process Data Segments (PDS's) and Known Segment Tables (KST's). The KST and PDS of the Initializer process, which performs Initialization (hence its name), are deciduous. The KST and PDS of any other process, and IOI workspace segments, are created in and of the storage system

hierarchy, and forcibly made a part of some process's hardcore environment. A forthcoming change to the online salvager output segment (>online_salvager_output) will similarly take it from the hierarchy, and place it in the global hardcore address space (by putting in the Initializer's descriptor segment during initialization). Such segments are known as reverse-deciduous.

- - - - - - - - - - - - - - - - - -

When one is to have paging, one must have a place on which to page. When Multics is booted, with the new storage system, the new supervisor may be being born into a world which has just seen a terrible crash. No VTOC entries can be trusted for sure. The maps of free storage on the disks may not have been updated. Hence, it cannot use these maps to determine where to page. The Physical Volume Salvager is capable of reconstructing these maps: he can determine what is real and what is not. But his services cannot be obtained until a place is found to page, for he cannot fit in main memory with all of the necessary support software. The old storage system salvager had a similar problem, and for him, a region of disk was defined known as the "salvager partition", where he would build his world, as he viewed the crashed Multics in a third-person sort of way. He could have all of the salvager partition to do whatever he wanted with- he didn't save anything there from salvager run to salvager run, and nobody else used it for anything, so he could totally ignore the previous contents, and set up an empty free storage map to describe it.

This policy has been adapted into the new storage system to provide a place for the supervisor to build his home where no one else can lay claim. A region of the Root Physical Volume (RPV) is reserved for this purpose:

> The hardcore partition is a region of the RPV designated by the RPV label, whose contents are defined as void upon bootload. The supervisor, upon bootload, can page there without fear of destroying anything.

The supervisor, when make_segs_paged is about to start his thing, sets up the hardcore partition of the RPV as the place from which all supervisor pages are drawn. Hence, the validity of any free store map on disk can be checked by the Physical Volume Salvager, who can now be operational before any is believed.

Unfortunately, the organization of new storage system segment control is such that only one free store map can be associated with a drive. Rather than declare the hardcore partition to be a separate "virtual drive", it was decided to use

the hardcore partition's free store map as the free store map for the RPV (as its segments, in point of fact, reside on the RPV) until the real map is checked out and installed, at which point the rightful free store map of the RPV takes its place.

There are some fairly subtle implications here. First of all, any pages of the hardcore supervisor which were not created before the rightful RPV volume map took its place (this event is known as the acceptance of the RPV (in Collection II)) will be withdrawn against the real RPV volume map. Such pages might be a region of the Initializer's stack that had not been reached before this, or perhaps the hardcore segment into which directories are listed when you list a directory via the list command. There are two implications of the withdrawal of hardcore pages against the real RPV volume map: 1) If the RPV volume map was depleted right before one of these supervisor withdraws, the supervisor would not be prepared to take out-of-physical-volume, or a segment move or failure thereof as an answer. Depending upon what segment encountered the page shortage, anything from a crawlout and online salvage (if a page of system_free_seg) to an undiagnosable, undumpable, un-ESD-able crash (page of initializer's PDS) might result. 2) If a page is taken from the RPV volume map, it had better be deposited at some time, or else the next bootload will find fewer pages available, the next fewer, etc., until there is no more RPV left. This unfortunate situation is known as RPV creepage.

The old storage system's solutions were simplistic. Again, since all pages came from one pool, out of disk for ANYBODY meant the end of the system's life. So that was no problem, so to speak (On the other hand, the big page pool was much bigger than one RPV). Not freeing the pages was more of a problem. As the old storage system supervisor shut down, it cut out its entrails, and then cut off its legs and then its arms, until only the knife was left. It burned all of its bridges behind it, depositing supervisor segments in a most careful manner as not to trip over itself. Not only was this very tricky, but was very hard to get right, and the old storage system often showed a creepage. Hence, the programs and data bases involved in the last phases of shutdown had to be either unpaged, or wired and paged, depending on the fact that allocations of pages were not performed until write time, and wired pages were never written. Hence, the obsolete concept of "wired_shutdown".

This was deemed to be a miserable problem, as it ensured that if any problem was encountered in shutdown, miscellaneous software would be missing, having been deposited, and emergency shutdown could not be tried (the so-called "repetitive ESD"), because programs that it called first were deleted. Therefore, the deletion of segments by shutdown was abolished.

When one abolishes the old order, one must put something in its place, and hence, the two problems above had to be solved. To ensure that pages for the supervisor would always be available, it was decided to assign them, via the nulled address mechanism described above, at bootload time. Supervisor segments are prewithdrawn:

> Prewithdrawing a segment consists of touching all of its pages, causing addresses (in the nulled state) to be assigned to all of them. Prewithdrawing may only be performed upon segments that cannot be deactivated, for were they are deactivated, addresses so assigned would be deposited. The conjunction of the AST bits DNZP (don't null zero page) and EHS (entry hold switch = don't deactivate) prevents the VTOC updater from noticing these nulled addresses. Prewithdrawn segments have a disk address associated with each page for the life of their activity (entire life for supervisor segments), and are thus not subject to out-of-physical-volume conditions.

All of the supervisor (with a couple of exceptions: see below) is prewithdrawn before the acceptance of the RPV. All process's PDS's (which are distributed parts of the supervisor in some sense) and KST's are prewithdrawn with the help of the segment mover before being put into service. This solves both of the above problems. First of all, the supervisor segments, so withdrawn, will live entirely in the hardcore partition. Since all addresses are assigned, and available for resurrection at any time, no pages will ever be withdrawn against the RPV volume map, eliminating potential creepage. Since the RPV volume map cannot therefore be encountered by the supervisor in a depleted state, we only have to worry about the supervisor depleting the hardcore partition at bootload time. If this happens, it will happen at such a time that the supervisor has not yet accepted the RPV, and hence, will not damage the hierarchy by its crashing. In this case, the hardcore partition is simply not large enough to hold the supervisor, and the RPV must be rebuilt (repartitioned).

An interesting corollary of this policy is that all of the deciduous segments live entirely in a region defined as being void at bootload. This is a very graphic analogy of the defoliation of flora: when a system is bootloaded, all of the deciduous segments find their leaves (pages) totally void and gone!

Since there is no creepage against the RPV volume map, nothing need be deleted at shutdown time, and emergency shutdown is always restartable and retryable. Try typing "ESD" after a successful shutdown to a 28-5 or later system. This is a major reliability feature.

However, there are a certain set of segments which are large, potentially not all used, and non-critical. In this class fall the system free segments (for listing directories), and some large workspaces used by the salvager. Were the supervisor to encounter an out-of-physical volume condition on these segments, it would be tolerable. Furthermore, prewithdrawing them would cause the hardcore partition to need to be larger than otherwise, taking space away from the RPV that might not be used. Futhermore, they are needed neither by normal nor emergency shutdown. In this case, the old policy seemed best. Thus,

> A delete-at-shutdown segment is a supervisor segment which is not prewithdrawn against the hardcore partition, not needed at shutdown, and deleted by emergency or regular shutdown. Such segments may have pages from both the hardcore partition and the normal multing region of the RPV.

Should emergency shutdown fail, and thus not delete these segments, it is a certainty that the failure of shutdown will be noticed by the next bootload, and a Root Physical Volume Salvage undertaken. This will collect all of these pages, and free them, as they appear in no VTOC entry on the RPV. (1)

_ - _ - _ - _ - _ - _ - _ - _ -

To make sure that the above policies are followed, the program that creates paged initialization segments (make_sdw, called by make_segs_paged and the collection II loader, among others) puts all segments into one of three categories:

> 1. Delete_at_shutdown. Marked as such in the SLT, by the bit slte.delete_at_shutdown, generated from the same as a keyword in the MST header. Only used for the few segments mentioned above. To make sure it gets deleted at shutdown, he puts it in a list of segments to get deleted at shutdown.
>
> 2. Explicitly managed (abs_seg). Marked by both zero cur length and max length, or slte.abs_seg, set from the abs_seg header keyword. You don't want this segment to be created routinely. It might be the pdmap_seg, where special bulk_store PTW addresses will be filled

---

(1) A Root Physical Volume Salvage is a Physical volume salvage of the Root Physical Volume (RPV). One can be initiated manually by the use of the "RPVS" parameter to the boot line. From the name of this parameter, the term "RPVS", generally heard as "Rehpoovis", has come to denote such a salvage. 28-5 and later systems do a RPVS upon every bootload after an ESD, to collect pages of descriptor segments, to prevent creepage.

in by init_pvt, or perhaps the salv_abs_seg's, which are not segments at all, but descriptor segment slots into which SDW's will be placed. To make sure that you don't try to use what make_sdw hands back, a null AST Entry pointer and zero SDW are returned. For many of these segments, the special entry make_sdw$unthreaded (please ignore the name) is called by the program which explicitly sets up the given segment, to make an ASTE.

3. Prewithdrawn. The default. Everything else is totally and personally prewithdrawn by make_sdw on the spot.

- - - - - - - - - - -