TO: Distribution

FROM: Steve Webber

DATE: 2/20/75

SUBJECT: Prelinking Overview

This MTB gives a description of the basic prelinking plan. There are several other MTB´s which also have a direct bearing on prelinking: these are an upcoming MTB on the proposed changes to the Standard Object Segment format, MTB-104 on the proposed changes to the KST structure, and MTB-154 on the proposed changes to the address space/name space managers in Multics. This MTB differs from some of the previous MTB´s as new ideas were incorporated into the design and other problems were solved.

> Nothing will ever be attempted if all possible objections must be first overcome.
>
> S. Johnson

Introduction

This MTB describes a proposed prelinking mechanism that would link many or all of the segments in the system libraries at system initialization time thereby avoiding, for the vast majority of cases, a great deal of duplicate work currently done by each process on the system. The changes to the system to implement the prelinking scheme are simple but extensive and include changes to:

1) the standard object segment format,

2) the KST and address space/name space management,

3) system initialization,

4) the PL/I compiler and the ALM assembler, and

5) new hardcore primitives for address space/name space functions.

All of these changes will be described in more detail below.

Note that within this MTB the mechanism of adding a segment to the address space of a process, i.e., assigning a segment number and filling in the KST entry, is called making a segment known. The function of mapping a reference name to a particular segment number is called initiating the name. Hence, segments are made known and made unknown and reference names are initiated and terminated. Initiating and terminating have nothing to do with the KST and segment number assignment. That is, they are entirely (user-ring) reference name table (RNT) functions.

The Basic Plan

The basic idea behind prelinking is to prelink as much as possible at system initialization time thereby avoiding duplicate work in each process. In order for such a scheme to work all processes that take advantage of the prelinking effort must reference the prelinked segments (and the associated linkage sections, etc.) with the same segment numbers much as all processes today reference all hardcore segments with the same segment numbers. As a means of convenience, this set of prelinked segments and related data bases will be referred to as <u>softcore</u> segments. The softcore segments, then, have permanent segment numbers for the duration of a given bootload.

There are several data bases which are used to describe the softcore segments in every process on the system. Some of the more important data bases are:

1. KSTT (Known Segment Table Template). The KST is a hardcore data base with ring brackets of (0, 0, 0) used primarily by the segment fault handler and maps the branch pointer and UID for each segment to a particular segment number. This data base has more or less the same structure as today´s KST with <u>all</u> name management removed. The prelinker generates a template KST which contains information for the softcore segments. This template KST is merged with the template_pds at prelinking time so that when a new process is created it initially has all of the softcore segments known.

2. SRNT (Softcore Reference Name Table). This data base contains the name management information used by the linker and name space manager (both removed from ring 0). The ring brackets on this segment are (1, 7, 7) and it is read-only after initialization. The segment lists all the reference names initiated for each softcore segment. The name space manager uses the SRNT to answer user-ring questions such as:

   A. What are the reference names for the segment whose number is <n>?

   B. What is the segment number of the segment whose reference name is <name>?

   The format of the SRNT will be similar to that of the URNT mentioned below.

3. TSS (Template Stack Segment). This segment is a template stack segment used during ring initialization. It contains a good deal of structure (all upward compatible with today´s stacks) including:

   A. a nearly completely initialized stack header,

   B. an initialized lot,

C. an initialized isot,

D. an optional combined linkage region (CLR) for the ring.

The possibility of having several TSS´s is being considered. This would allow different rings or different subsystems to special case their needs by placing the appropriate linkage information in the stack and thereby avoid the need for a seperate combined linkage segment altogether.

4. SCLS´s (Softcore Combined Linkage Segments). These segments contain the softcore linkage information. They are shared by all users (read-only with ring brackets (1, 7, 7)) and are copied into a user´s private address space (process directory) when they can no longer be shared. (See later)

5. CSST´s (Combined Static Segment Templates). These data bases contain the internal static data for some of the softcore segments. They have ring brackets of (1, 7, 7) and get copied into a process´s process directory when necessary. (See later)

The set of softcore procedure segments is determined by the administrators of each particular site. It would presumably contain nearly all of the segments in the system libraries as well as any special subsystems the site administrators might think appropriate. The actual mechanism to define the set is with <u>online</u> ASCII files (called prelinker driving tables, PLDT´s) describing those segments to be included and the reference names associated with them. The format of a PLDT is simple and it is possible to control in which (softcore) combined linkage segments the linkage for each softcore segment is to reside. (The ASCII file is converted into a binary equivalent which is actually used during initialization.)

The softcore segments, then, are defined by online, editable files which can be altered, converted to binary and "installed" any time prior to the bootload in which they are to take effect. The actual prelinking programs search a system directory (probably >system_control_1) at initialization time for any PLDT´s and prelink any segments specified therein.

Note that the reference names in the PLDT´s define which names should be initiated for a softcore segment. There is no need to have these names on the actual branches in the directories. This relieves directory hash table space. Note also that system library contents are defined by the PLDT´s (if so desired) and that it is therefore easy to make online installation of multiple segment subsystems (like PL/I) in a consistent way. The new versions will not be (implicitly) used by anyone until the next bootload.

There are several per-process and per-ring data bases that parallel those described above. These are:

1. URNT (User Reference Name Table). This is a per-ring user-writeable data base managed by the name space manager. It contains the same information about segments made known by users as the SRNT contains about softcore segments. In addition, the URNT contains the search rules for the given ring as well as the working directory for the ring. Note that the URNT and the SRNT are logically combined for a given ring to define the (reference) name space for the ring.

2. STACKS Each ring (1-7) has a stack which is initialized by copying in a template stack segment (TSS) and setting any per-ring pointers appropriately. Since the stack template defines a fairly complete working environment, little more than this copy need be done to provide a user with a "new ring".

3. UCLS's (User Combined Linkage Segments). Linkage information specific to a given user's process will be copied into the user's current combined linkage region (initially in the stack). As more combined linkage regions are needed, UCLS's are created in the process directory.

4. UCCS's (User Combined Static Segments). These segments are the image of the CSST segments created by the prelinker. They are placed in the process directory (automatically) when needed. (See later)

The search rules will be functionally equivalent to what we have today, namely a list of absolute pathnames interspersed with several keywords. Although libraries in the prelinked set may appear as absolute pathnames in the search rules, the searching of the actual directories need not be done each time a search is performed. Details of the working of the search mechanism is given later.

Advantages/Disadvantages

The advantages of prelinking fall into two basic classes:

1. less execution time required in a process,

2. fewer page faults required in a process.

The execution time gains are primarily in the realm of linking where many fewer linkage faults will occur and those that do occur will be resolved faster. Faster ring initialization will also make multiple ring functions more efficient and hence that much more attractive to use.

Paging gains come from several areas but are concentrated in the increase in shared data available under the prelinking scheme. Most of the combined linkage and name space management segments of a process will be shared with all users. Second order effects will also be great as fewer pages being actively used will be displaced by directories and the like that would have to be brought in if complete dynamic linking were required.

Other advantages of prelinking, which do not have a direct bearing on performance, are:

1. all processes share more segment numbers thereby making debugging and the like easier,

2. logins and new_procs will be faster, but likely replaceable in many cases by new_ring and new_stack.

There are many disadvantages to undertaking the prelinking project, but most of these are not related to the end product but rather what must be done to get there. In particular, the following problems are only temporary and will not exist after the conversion:

1. the standard object segment must be respecified (all users of object_info_ must change),

2. the PL/I compiler must be changed,

3. the binder must be changed,

4. the ALM assembler must be changed,

5. much documentation must be (re)written including:

   a. the MPM sections on standarrd object segments,

   b. the PLM´s describing the actual implementation,

   c. user info segments describing any user-visible changes,

   d. the MPM sections on the PL/I compiler and ALM assembler,

   e. the MPM sections on all the new interfaces,

   f. the MPM sections describing reference names, searching, and the linker,

   g. MTB and MTR documentation as the project proceeds, and

h. SRB updates.

6. the system primitives for directory manipulating must be redesigned and rewritten -- write-arounds must be provided for the current versions,

7. new user-ring interfaces will need to be designed which better interface the new hardcore primitives,

8. users and system programmers will have to modify their idea of what the standard programming environment is -- although it will not change functionally very much,

9. there are many other changes being made to the system which will have to be coordinated with the prelinking effort.

System Initialization

The changes to initialization to provide prelinking are isolated in two areas, the making known of directories and segments and the prelinking itself. The end result after initialization would ideally be the creation and initialization of the following data bases:

1. Template PDS including template KST
2. System RNT
3. Combined Linkage Segment(s) (CLS´s)
4. Combined Static Segment(s) (CSS´s)
5. Template Stack Segment(s) (TSS´s)

The template KST and SRNT would include all prelinked segments, the 5+ segments above, and all parent directories of any prelinked segments. The only problem introduced here is that the initializer or bootload process must make several directories and segments known before prelinking can be undertaken. The plan is to perform this in a non-standard, temporary fashion until prelinking does it for real. Once prelinking is complete all references to any of the directories or segments referenced during initialization will be through the softcore segment number mapping generated during prelinking. Note that we do not want to initiate any reference names in ring 0 (other than those prelinked) due to the upcoming changes to the address/name space management routines.

The initialization will therefore work nearly as today until we finish reading in collection 3 (the bulk of the contents of >system_library_1). When these segments are all read in the prelinker is called to do the following:

1. Stage 1 - locate all PLDT´s and set up for scanning them. Create all the segments to be initialized (SRNT, SCLS´s, etc.).

2. PASS 1 - a first pass is made over all the PLDT´s. Each segment indicated is made known and the specified reference names are initiated. The segment numbers assigned are bound to the reference names for the life of the bootload.

3. PASS 2 - a second pass is done over all the segments indicated in the PLDT´s. This pass examines each link pair in each linkage section and snaps each link it can using the reference names initiated in the first pass. The search rules used if more than one segment having the same reference name are specified in one of the PLDT´s.

4. PASS 3 - this is an optimizing pass done to reorder the SRNT to minimize paging on subsequent searches caused by user linkage faults. This pass is unnecessary but likely to be beneficial.

5. STAGE 5 - Cleanup the various data bases and perform any further initialization (such as filling in IOCB´s, etc.) that is appropriate for nearly all processes.

Several side effects of the above work might well be noted here. First, during PASS 2, when all definition sections are searched, the SRNT entries are filled in so that the value of the offset of any entry whose name is the same as a reference name on the segment is saved with the reference name. This means that any references of the form a$a can be satisfied completely with the SRNT search. Since such references are by far the most frequent many linkage faults will be satisfied that much faster. Further, since nearly all commands fall into this class, we can probably eliminate the command processor associative memory which requires "back door" entries for clearing it when reference names are terminated.

The dynamic linker will be changed to take advantage of this mechanism thereby avoiding the definition searches as well as the directory searches for a large number of linkage faults.

A second side effect of the prelinking phase is the ability to provide limited metering of entries in the softcore set of programs. The PLDT´s could specify certain entries which are to be metered. These entries will be invoked only after an "aos" counter in an inner ring is updated. Since all softcore segments will always be referenced by their softcore segment numbers and since their entries will therefore remain "constant" we can know that the meter reflects the actual number of calls to the entry through the prelinked links. (See later description of "Entry Meters".)

The Issue of the User-ring Linker

Most people agree and it has been accepted by Multics designers that it would be best to remove the linker from the hardcore supervisor -- ring 0. Whether or not it is moved to the user ring or only to ring 1 or 2 is an ongoing debate that fortunately need not be answered in order to implement prelinking. The linker outside of ring 0 is indeed easier to implement with prelinking in effect because:

1. the bootstrap problem is solved,

2. any performance problems are minimized because of the drastic decrease in linkage faults and the increased efficiency of the hcs_/real_hcs_ write around scheme, it having been prelinked.

The reason the user-ring linker is mentioned in this document is because of its interaction in the overall installation plan. By the time it is installed, prelinking will already have been installed and hence the new features will be available. It will not be installed with the first stages of prelinking so that we can isolate each stage in the overall installation plan.

The Issue of the User-ring Name Space Manager

The user-ring name space manager is similar to the user-ring linker in many respects having to do with installation plans. It, too, will not be installed in its final form until after prelinking has been. However, a hardcore version of the new name space manager will be installed with the first prelinking installation. This will make removal of the mechanism to an outer ring easier and give us time to begin reprogramming for the new primitives the user-ring name space manager requires. The eventual removal of the name space manager from ring 0 should coincide with the removal of the linker from ring 0.

The shipment to other sites (than MIT) of the entire system including prelinking, the user-ring linker and the user-ring name space manager need not be done all at once.

In that prelinking will depend on the separation of the RNT from the KST the final versions of both of these should be used from the start.

Prelinker Driving Tables (PLDT's)

The PLDT's are the main driving tables for the prelinking task. The tables describe exactly which segments should be made known and exactly which reference names should be initiated for each. In addition, they describe in which combined linkage segments the linkage for a softcore segment is allocated as well as in which combined static segments the static should be allocated. The PLDT's also include search rules and metering information to be used during the prelinking task.

If no PLDT's are found, the system works basically as today.

The PLDT´s are originally ASCII files in a format similar to an MST header or a bind file. These ASCII files are "compiled" into binary format (prior to the bootload which uses them) for efficiency reasons.

The format of a PLDT is as follows:

1. There are 4 major keywords which are:

   linkage,
   directory,
   segment, and
   search_rules.

2. For the "segment" keyword there are 2 minor keywords which are:

   refname, and
   meter.

(A "static" keyword is being considered. It would allow the PLDT creator to specify where static storage which must be preallocated is placed (see later).)

The "linkage" keyword instructs the prelinker to place all linkage sections of the segments following into the specified segment(s). All linkage is placed in this set of segments until another linkage keyword is encountered. The format of the linkage statement is as follows:

```
          linkage:     name[,size];
```

where name is the first component of the name of the combined linkage segments created. A numeric suffix is appended to the name as new SCLS´s are created. The size field is a decimal integer (default 64) which indicates the maximum size of any of the SCLS´s of this class. For example, if there are 20 records of linkage information and the line

```
          linkage:     sss_linkage, 16;
```

appeared, two segments named

```
          sss_linkage.0       (16K)
          sss_linkage.1        (4K)
```

would probably be created. If the name includes the string "stack", the linkage is placed in a segment in a format consistent with the segment being a stack, i.e., one of possibly several TSS´s being created.

The "directory" keyword instructs the prelinker to search the given directory for any segments encountered with a subsequent "segment" keyword. A subsequent "directory" keyword will cause a new directory to be searched. The format of the directory statement is:

```
          directory:     dirname[,dirnamei]...[,-all];
```

where dirname is the name of the directory to search and dirnamei are synonyms of the directory (used in setting up search rules).

If the "-all" option is specified, the prelinker and name space manager can assume that all reference names associated with segments in the directory have been specified in the PLDT´s. This means that the directory itself need never be searched during normal running of a process. Such a directory is said to be completely prelinked.

The "segment" keyword must be preceded at some time by a "directory" keyword and specifies the name of a segment to prelink or, if the segment is not an object segment, a segment that can at least be linked to. The format of the segment statement is:

```
          segment:     segname;
             .
             .
             .
          end;
```

After the "segment" line and before the "end" line must appear at least one "refname" line and can appear several "meter" lines. The refname lines are structured as follows:

```
          refname:     name[,namei]...;
```

where name and namei are reference names applied to the given segment.

The "meter" keyword is used as follows:

```
          meter:     entrypoint[,entrypointi]...;
```

and specifies those entrypoints to be metered.

The format of the "search_rules" statement is:

```
          search_rules;
             path1;
             path2;
                .
                .
                .
             pathn;
          end;
```

and specifies the order in which directories are searched during prelinking. If no "search_rules" statement is encountered in any PLDT the default search rules are simply the order in which "directory" statements are encountered. If there are more than one "search_rules" statements, the last encountered is used. Prelinking search rules are only appropriate if duplicate reference names are encountered. Note further that a "referencing_dir" search rule is always the implicit first search rule during prelinking.

The structure of the PLDT´s has been given in detail because it is the most important externally visible administrative interface to the prelinking mechanism.

PROBLEMS WITH PRELINKING

The prelinking design had to overcome many problems to reach a workable, useful structure. Some of these problems require widespread changes to the system for solution but none of them appear to have any unrealistic requirements. These problems and the proposed solutions are given in the following sections.

The Problem of Static Storage

One of the goals of prelinking is to share much more of the system thereby minimizing working sets for all processes. The prime target of sharing is, as might be expected, the data bases of the linker and the linker´s output the linkage sections. Unfortunately, internal static storage is currently allocated in the linkage section. This mixes shareable data, the snapped links, with unshareable data, the per-process (and per-ring) static storage. To solve this, it is proposed that internal static storage not necessarily be placed in the linkage section but rather in a section of its own. This has widespread implications throughout the system. First, it means the standard object segment format must change to provide for another section (resulting in text, defs, links, static, symbol, and map). Second, it means that translators like the PL/I compiler and ALM assembler must use different accessing code when referencing static. Third, it means that all users of object_info_ and the like must be changed to expect potentially new object segment information. Fourth, it means changing the binder and friends to handle the new cases. None of these requirements is prohibitive especially since all can be done without "flag days".

Removing the static from the linkage for a segment only solves some of the problems, however. Another problem arises when we want to snap a link to a location in static at prelinking time. It was the original intent to allocate storage in the combined linkage for a ring (not the shared SCLS´s) for static and copy the static from the object segment into the allocated storage, both of these happening at first reference to the static in a ring. This, however, makes it impossible for the prelinker to resolve a link pointing into a segment´s static because neither the segment number nor the offset within the segment of the static region can be known. To solve this problem the following action is taken:

A. If an object segment has definitions into its static, that static is copied into a (softcore) combined static segment at prelinking time.

B. If there are no definitions into an object segment's static, the static is not copied at prelinking time and only copied later on if reference is made to the static (by the procedure itself).

Note that the static for an object segment may be allocated in the linkage section as is the case today. Hence, in this case, when we say the static is placed in a combined static segment the linkage is placed there as well.

This scheme allows us to snap links to static data areas at prelinking time since we know the segment number and offset of any static region that has definitions pointing into it. It does mean, however, that since we want to conserve segment numbers wherever possible, it would be more convenient to allocate all such static regions in as few segments as possible and that a user forced to use these segments would therefore get in his process (ring) a copy of an entire combined static segment. Although in some cases this may increase the working set of the process, these cases are few and the working set would only be marginally increased (it depends on breakage of preallocated static regions).

So we see the prelinker has the task of preallocating some static regions as well as all the linkage sections.

It should probably be noted here that currently there is much more internal static storage being used than need be. This is usually because internal static is a convenient place to store named constants which are not part of the PL/I language. If named constants were placed in the text section and did not have to be copied when passed as procedure arguments it is estimated that more than half of the internal static in the system libraries could be eliminated. This means that much more data can be shared and the working sets would be that much smaller. This is currently being considered as an extension (optimization) of our PL/I compiler.

In general, the system would run better if large regions of internal static storage that were not initialized were allocated explicitly so that the actual object segments and copied static regions for bound segments would be smaller. (It has been an unfortunate consequence of our current binding strategy that referencing one component of a bound segment brought in the entire static for the bound segment including often large regions never used by the process but that added to the process's working set.)

A different kind of problem arises with the use of external static storage. In particular, type 6 links (create if not found) pose a problem to the prelinker. The prelinker could create softcore segments for any such links and thereby resolve the links. Rather than do this, however, it is proposed that the use of such links causing implicit segment creation, etc., be forbidden for system programs as it may interact with a user´s process in strange, unpredictable ways. This includes external static placed in stat_, the default target for external static variables. Similarly, the technique of renaming stat_ should also be prohibited. The equivalent can be achieved through more direct means with more efficiency. It is thus proposed that a certain class of segments be unacceptable candidates for prelinking and that the prelinker refuse to create any segments as a result of a type 6 link. (If such segments did get placed in the prelinked set, things would still work but the linkage segment created by the prelinker would contain unsnapped links. To snap these links the entire linkage segment would have to be copied -- see later.)

The prelinker will generate a listing of all errors and links it could not snap as these can be an unexpected cause of a decrease in system performance (when linkage segments get copied -- see later).

Unsnapping Links

Besides static storage, the prime reason a linkage section is modified today (other than by the linker) is to unsnap links. This means replacing the snapped link (an ITS pair) with its original fault tag 2 and data for the linker. The difficulty this poses with respect to prelinking is that most links including some that may want to be unsnapped are in the system-wide, shared, read only combined linkage segments.

The proposed solution to this problem can actually be used to solve several other similar problems with softcore segments. The solution is a new feature in the system acting primarily on (but not necessarily limited to) softcore segments. It is a "copy-on-write" mechanism for the entire segment. That is, any attempted modification of a read only segment is intercepted and, if appropriate, a copy of the segment is placed in the process directory. In addition, the new copy is made known with the same segment number that the original had. By applying this technique to the combined linkage segments the first attempt to reset a snapped link will cause a copy of the entire segment (into the process directory). From then on the user has full freedom to modify the copy. Although the user has now lost the protection of a read-only linkage segment and the benefits of a shared linkage segment he has retained the advantage of having most of the links in the copy still snapped. He has also potentially increased his working set to larger than it is today (by having linkage sections that he normally wouldn´t have in his combined linkage). But the user who wants to make system segments unknown must pay the price. He has surely gained in the long run.

This copy-on-write feature could also be used for the combined static segments as well. This avoids copying until the static is actually modified. Similarly the segments free_ and tree_ which currently achieve the same end via the copy switch could use this alternate scheme. Indeed, it has been proposed that copy-on-write replace the copy switch. Instead of this, I propose we use the copy switch to tell us when copy-on-write is allowed.

The manner that copy-on-write would then be implemented is as follows:

A. if a no_write_permission fault occurs, see if the segment has the copy switch ON. If not, signal "no_write_permission".

B. if the copy switch is ON, create a copy of the segment in the process directory, make the original segment unknown, and make the copy known with the original segment number. The access on the copy will be set to "rew". Note that all of these actions are on the KST not the RNT. All reference names that were originally there remain. Note further that this entire mechanism can be implemented in the user ring although some optimization provided by the supervisor may be valuable.

There are currently several problems with the use of the copy switch. In particular the definition of what it does may not be the most appropriate. Do we want a copy each time we explicitly make a segment known even if the segment is known or is a segment which is a copy of another... The copy-on-write replacement seems to clarify these issues as well as providing the interface that is apparently desired. Each attempt to modify the original will get a new copy -- attempts to modify a copy will only get a new copy if the access to the copy has been set (back) to prevent modification and the copy switch has been set ON for the copy. Note that if the original segment allows modification no copy will be created even if the copy switch is ON. This may lead to difficulty, but the setcopysw command could issue a warning if a non-SysDaemon user has write permission to the segment. Similarly the set_acl command could issue a warning if write permission is being given to a segment with the copy switch ON.

Another interesting feature of the copy-on-write mechanism is illustrated in the following scenario. Suppose a user makes a potentially copiable segment known and saves the pointer to it to be used later in order to delete the segment. If subsequently an attempt to write into the segment is made a copy will be created. The attempt to delete the original via the saved pointer will instead delete the copy. Users making use of the copy-on-write and copy switch mechanisms had better understand what they are doing.

For added consistency, it is proposed that the segment truncation and deletion primitives of the system be modified as follows:

1. An attempt to delete a segment with the copy switch ON will fail in the same way a segment to be deleted with the safety switch ON fails.

2. An attempt to truncate a segment with the copy switch ON will cause the copy-on-write mechanism to be invoked. The copy created is then truncated.

These two changes need not be implemented in ring 0 and are intended to be protection against unexpected use of the copy-on-write feature.

It is now clear why several combined linkage segments can be created by the prelinker. By using several segments, it may not be necessary to copy as many pages if some links are unsnapped thus preserving as much sharing as possible. It is also now clear how we can recover from unsnapped links in the prelinked combined linkage segments. When the links are finally snapped (if ever) a copy of the combined linkage segment is created automatically. The (user-ring) linker need know nothing about it.

Search Rules and Reference Names

One by-product of prelinking is a completely initialized SRNT giving the mapping between segment numbers and reference names for softcore segments. When a new ring is initialized these reference names are not initiated. Instead, the initiation of these softcore names occurs on first reference (i.e. as a result of something invoking the search rules and locating softcore segments) except for any inter-softcore seg references which have, of course, already been resolved using the mapping in the SRNT. The basic change from today is that the entire system has effectively been bound together as a unit instead of the smaller entities of today, bound segments. It is not possible today to remove the internal bindings the binder has generated nor is it reasonable for a user to assume knowledge of which system segments are bound with which. For these reasons, it is not considered an undue hardship that the internal bindings generated by the prelinker are in effect at the start of execution within a ring. An example might clarify the change that the prelinking scheme will cause. Consider a process today that references a private version of the print command before the system version is referenced. This will cause the reference name "print" to be associated with the private command and hence any system code _not bound in with the system print command_ will invoke the user's version (assuming somewhat standard search rules). With the prelinking system all system code will reference the system version of print unless that binding is explicitly removed (say, with the tmsr command). This is true even if the first reference to print in the ring was not to the system version. This is an incompatible change and, although few users would notice and even fewer would care, it may be a hard issue of which to convince other users.

Assuming this change is acceptable, the rest of the reference name structure will be described. At prelinking time each reference name initiated is assigned a unique integer. This integer, the reference name index (rnx), is used as an index into a per-ring bit table describing which of the system reference names have actually been initiated in the given ring. Initially this table indicates that no names are initiated. Whenever the ring finds a softcore segment by name (as a result of searching), the associated bit in the reference name bit array is set. This effectively initiates the reference name in this ring. To terminate a system reference name, this bit is turned OFF (if it is ON) and any references to the segment associated with the name, in the system as well as the per-ring linkage sections, are removed.

Note that terminating a softcore reference name should probably unsnap the links to it therefore causing copies of linkage segments to be created. In fact, there should probably be two seperate primitives for terminating reference names of softcore segments. The first should unsnap all links while the second, and probably more frequently used, should merely turn off the bit in the per-ring bit table.

The search mechanism usually works as follows: first, the SRNT is searched for the name, then the per-ring bit table is examined to determine if the name has been initiated. If so the search terminates. If not, the work of the lookup in the SRNT is remembered and the URNT is searched. Again, if the search is successful the search terminates. If the name has not yet been initiated in the ring, the other directories specified in the search rules are searched. If the name is found in one of the system directories specified in the user's search rules, the corresponding bit is turned ON in the bit array and the earlier-saved information is returned as the result of the search.

Note that no system directories are ever actually searched (if they have been completely prelinked) and the system reference name table is searched only once.

The users will specify search rules exactly as today. Any pathnames encountered that represent completely prelinked directories will be encoded in the actual data base representing the search rules so that the correspondence can easily be determined at search time. In particular, if a directory is prelinked that is not specified in a user's search rules, the reference names of segments in that directory will not be considered during the search.

It is currently planned to place the bit array describing which system reference names have been initiated in the URNT (a per-ring data base).

It is considered legal for several prelinked segments in seperate directories to have the same reference name (although this will be rare). When this happens, the user´s search rules will select the appropriate one.

Pathnames as Reference Names

In the current system the hardcore uses reference names in a strange and erroneous way. In particular when a directory is made known, its pathname is placed in the KST as a reference name so that the routine find_ need not always recurse back to the root directory when trying to resolve a pathname. Instead, find_ stops when it notices a given prefix of the pathname it is trying to resolve has already been initiated. The flaw with this scheme is that renaming a directory or removing a name from a directory that has been made known does not purge the KST´s of no longer valid pathname-segment number associations.

Several solutions to this problem have been proposed including, 1) establishing an elaborate "trailer" mechanism so all KST´s that contain a given pathname can be located, 2) not allowing renaming of directories (or delay the effect until offline salvage time), 3) managing a system wide data base of pathnames that can be mapped easily onto a user´s name space, and 4) not allowing find_ to use a pathname-segment number association.

It is proposed that this problem eventually be solved as part of the prelinking project. The cleanest solution seems to be to recurse back to the root in the user-ring version of find_. However, due to the potentially long conversion period when pathnames will be used by commands, there may be a large overhead in solving this problem with the first versions of the prelinker. Metering tests are being run to determine the actual cost of doing this; the decision of whether the initial prelinking system fixes the bug will be resolved after analysis of this data.

If it turns out that it would be valuable to optimize the find_ function (and its inverse hcs_$fs_get_path_name), a per-ring table of pathnames managed by find_ could be maiNtained in a small associative memory. This solution, of course, does not fix the bug, but it does provide an efficient mechanism in the interim before the bug is finally fixed, i.e., after enough of the system has been converted to use the new interfaces efficiently.

It may be noted here that, due to a bug in the system, for several years find_ actually did recurse back to the root. When the recursion bug was "fixed" few users noted any improvement in the system.

Process Creation

The process creation task will not differ much from what is done today, but the end result will be a process much further along in its own initialization. The answering service will perform the following steps:

1) create a process directory

2) create a "pds" in that directory

3) initialize the pds by first copying in the template_pds (including the entire softcore KST in an initialized state) and then filling in per-process variables

4) create a "pit" segment and initialize it by first copying the template_pit. Note that action will be taken to make the pit known in the new process´s KST with a softcore segment number.

5) allocate and fill in an APT entry.

6) tell the traffic controller to start the process running.

The KST which is contained in the PDS is in a completely initialized state. In particular all softcore segments have been made known (their UID and branch pointers filled in) and most already have the access control information filled in. (When the prelinker runs, it fills in the access control information for any segments which have only *.*.* on the ACL.) The DTBM is also set appropriately to 1) 0 if the access control information is not filled in, and 2) the value at the time of prelinking otherwise. This means that selective access on all softcore segments is still possible and that access on softcore segments can be changed at any time in the normal way.

The Internal Static Offset Table

The internal static offset table (isot) is a table used by a procedure to find a pointer to its internal static storage much the way the lot is used to find a pointer to the linkage. In particular, a pointer to the isot for a ring exists in the stack header and points to an array of packed pointers, indexed by procedure segment number, to the static regions. The isot entry for a procedure is initially set to a value that will cause a fault if an attempt to load it is made. The handler for this fault (which happens once per segment per ring) allocates storage for the segment´s static in the combined linkage segment and then copies the static from the object segment into the UCLS. Finally, the isot pointer is filled in to point to this dynamically allocated static.

The first reference to the isot entry (the one causing the fault) in a ring is usually in the entry operator for the given program. This is because PL/I programs save linkage and static pointers in their stack for efficiency. An option is being considered to bypass the fault mentioned above by explicitly examining the isot value and taking appropriate action if the static is not allocated. This would be done in the (new) entry operator for PL/I programs and is hence independent of any compiled code.

Ring Initialization

A ring is initialized by the system at first reference to it. The most common cases are 1) on an outward call from ring 0, and 2) on reference to an inner-ring gate procedure. The first reference is trapped by the system and the ring is initialized by the (system) handler for the outward call condition. The second case causes initialization of the ring by the (system) segment fault handler in response to a segment fault on the stack segment for the inner ring.

The actual work required to set up a ring consists of the following steps:

1) create a stack segment in the process directory,

2) copy the stack template into this new segment,

3) initialize the per-ring variables in the stack header, and

4) if a combined linkage segment is required (i.e., the linkage is not in the stack) create and initialize the first combined linkage segment for the ring.

The stack template copied in contains an initialized linkage offset table (lot) as well as an initialized internal static offset table (isot). All lot entries for softcore segments have been filled in to point to the linkage for the respective softcore segment in one of the shared, softcore combined linkage segments (or the stack template). The isot entries have also all been filled in with either:

1) a value that will cause a fault if an attempt is made to use it, or

2) a pointer into the combined static segment for softcore segments which have definitions into their static.

Note that any linkage in the stack segment must have the corresponding lot or isot pointers changed at ring initialization time as the segment number of the stack segment is a function of the ring being initialized.

The stack header is entirely filled in by the prelinker except for those pointers into the stack itself which are per-ring. These latter pointers are filled in as part of ring initialization in the same way the lot and isot pointers for segments whose linkage is in the stack. Note that no segments need to be made known or reference names initiated to fill in the various operator pointers, signal_ pointer or unwinder_ pointer.

It may turn out to be convenient to combine the steps of ring initialization into a "new_ring" command. This command could leave everything as it stands in the process directory (except possibly the names on appropriate stack and linkage segments) and create a new, fresh execution environment consisting of fresh static storage and a new stack. This command would probably be used as a replacement for the new_proc command in many cases where it is desired to retain, for debugging, the state of the process.

PL/I Changes

There are several changes required of the PL/I compiler before we can take full advantage of the prelinker. Some of these are optimizations which are independent of prelinking but which become more important because of it, while others are changes that the prelinker will depend upon. In particular, the following "optimizations" are being considered by the language group:

1) restructuring the trace command (compatibly) so that internal static storage is not allocated at compile time but rather storage is allocated when and if a particular entry is traced,

2) introduction of the "options (constant)" attribute for internal static storage variables. This would merely be an implementation optimization, not a change to the language, which would allow variables (named constants) to be placed in the text section even though we pass them by reference (i.e., without a copy). Passing variables by reference in this case means the possibility exists for the called procedure to <u>set</u> the variable -- a common means of destroying constants in other operating systems. With Multics one can avoid most problems of this nature by making executable code read-only, as is the normal case.

3) Recoding several programs of the compiler itself so that type 6 (create if not found) links are used in an acceptible way.

4) changing the compiler to optimize the use of "search" and "verify" tables. These tables are currently allocated in the text of the procedure using them. It is proposed that the code generator optimize the more commonly used tables by sharing them -- somewhere in pl1_operators_. The actual number of canned-in tables would probably not exceed 10 or so.

The following changes to the PL/I compiler are more than optimizations and until a program is compiled with a version of the compiler having these features, full utilization of the prelinking of the programs will not be achieved. These changes include:

1) changing the code generator of the compiler so that internal static references and linkage references are not necessarily based on the same pointer value, because the two regions may no longer share the same virtual memory.

2) changing the object segment generation components of the compiler to make a new-style object segment, i.e., one with a separate static region.

Due to code optimization assumptions in some programs on the system (including the PL/I compiler itself) and due to the requirements of the overall installation plan discussed later, the new PL/I compiler will create an object segment with static in the linkage section by default, i.e., as is done today. A new control argument (that users needn't know about) will instruct the compiler to generate an object segment with the linkage and static separate.

Another change to the PL/I compiler being considered consists of the redefinition of where entry descriptor pointers are placed in an object segment and how they are found. The intent is to use a bit in the flags word of the entry sequence to indicate the new referencing scheme for the entry descriptor pointers. This would allow the compiler to generate the entire entry descriptor data structures in the text section of the object segment and thereby avoid a good deal of needless overhead when trying to find the entry descriptors. This change is mentioned here because with the prelinking project comes the assumption that we will recompile the system (or at least that portion we want prelinked) and that this recompilation should be done with a version of the PL/I compiler which has the necessary features. This latter change is actually needed by the project of extending and optimizing the command processor.

ALM Changes

As with the PL/I compiler, the ALM assembler must be changed to generate new object segments and recognize the new "static" component. In particular, this means the "join" pseudo-op must be extended to handle multiple (at least 1) location counters in the static region.

Again, as with the PL/I compiler, this is as good a time as any (?) to add two other features to ALM which have little bearing on the prelinking project but which would be convenient. One, and the more important, is an "entrybound" pseudo-op which allows the programmer to instruct the assembler to save the given value as the entry bound for the segment. This entry bound will be placed in the object map for the segment and gives us a better handle for manipulating gate segments.

One last change to ALM is a proposed "text-embedded-link" feature which would be used primarily during system initialization but has a potential for user-ring use once we understand better how to manage the "permanent pointers" that prelinking provides. These text-embedded-links would be used during system initialization to avoid the clumsy initialization of ITS pointers in many hardcore procedure segments (e.g., fault and interrupt handlers). These links could also be used during prelinking to avoid use of linkage sections completely for the prelinked segments (as well as preventing the unlinking of them). Although there is no immediate plan to use text-embedded-links anywhere except during initialization, the addition of the capability to handle this feature in the assembler would be convenient and allow further research.

The Storage System Associative Memory

Several months ago, MTB-104 discussed a change to the structure of the KST that is, as stated then, required for prelinking. Among the features proposed was a scheme whereby access calculation could be avoided in many cases by keeping the date-time-branch-modified (DTBM) value for a segment in that segment´s KST entry. Comparing the value of the DTBM in the KST entry with the value in the branch allows us to verify whether the data we placed in the KST entry is still valid. The prime unfortunate side-effect of this scheme is that the check (of UID and DTBM) still causes a reference to the branch of the segment. (Today we not only make this reference, but we also calculate the access from scratch each time...) It is proposed that this problem be solved with a small associative memory of storage system information. This storage system associative memory (SSAM) would contain the UID of a segment (or directory) as well as its DTBM. In addition, it could contain other information such as bit count, primary name, etc. The intent is to minimize paging by concentrating the most referenced data in pages more likely to be in main memory.

The SSAM would be organized by a simple hash scheme based on a segment´s UID. The SSAM can be managed (read) without a global lock associated with it. This means that it is easy to validate the contents of a KST entry as well as to render obsolete the contents of the SSAM entry for a particular segment when that segment´s branch is changed. The SSAM will thus be assumed to be a part of the new KST structure and hence required for prelinking.

The actual implementation of the SSAM being contemplated is to merge it with the Active Segment Table (AST). (The UID hash mechanism within the AST is needed by the new Storage System.) This means that the reference to the branch will still occur if the segment itself is not active, but this is not too bad since the segment will most likely be activated before the page is evicted from main memory and the page will be needed for activation anyway -- hence, still no unnecessary paging.

Prelinking Installation Plan

Installation of the prelinking extensions to the system will come in several phases. The earliest installations will not require any of the compiler, assembler, or binder changes, but will require the reworking of the name/address space managers -- i.e., splitting them apart. Later installations will take advantage of the compiler (etc.) changes by taking different actions on a segment based on which version of the compiler (binder, etc.) created it. In time, all of the prelinked segments will have been recompiled and the prelinker can then take maximum advantage of all system changes.

The first installation stage will consist of a system with none of the linkage, static, etc. segments being shared. Instead, process or ring initialization will explicitly copy all of these segments into the user's process directory. The size of the segments copied can be limited during this phase by limiting the number of segments prelinked. The copy-on-write feature is thus not required for this stage.

The second stage of installation also precedes the new compilers and places as many linkage sections as possible into shareable combined linkage segments. This includes all linkage sections with no static and those linkage sections whose static consists entirely of words reserved for the (by then) obsolete trace mechanism. This stage will require the copy-on-write feature in order to unsnap links in these shared combined linkage segments.

Before new object segments can begin to appear in the system libraries, management of the isot must be provided. This means that the prelinker must generate the isot and the various operator and object segments must use it.

As these first two stages unfold, much user documentation will have been prepared and released describing the upcoming changes to the standard object segment format and the programming environment in general.

The next major phase in the installation plan is installation of all the necessary system programs to work with the new object segment structure and the new interface to object_info_. Once these programs (about 20) have been installed, the binder, object_info_, get_bound_seg_info_ and the like can be installed. This precedes the next stage which is the actual installation of the PL/I compiler and the ALM assembler which generate the new object segments. (The compiler and the binder were, of course, both necessary to check out each other.)

From this point on, the more of the system that is recompiled and rebound the better the system will run. Although there is no date before which the entire system must be recompiled, it can only help to do so. During this stage of running the system, new object segments will have their linkage copied into the shared combined linkage segments while the static won´t be copied at all unless there are definitions into it. In this case the static is copied into the (softcore, but non-shared) combined static segment(s) which get copied-on-write into a process´s process directory if referenced. It is therefore advantageous to get as many programs reprogrammed as soon as possible in order to minimize the size of (per-ring) combined static segments.

The advantages of requiring the recompiling of the system are many. One disadvantage, in a sense, is the urge to reprogram as well. In fact, there are several simple reprogramming changes which could be done as part of the recompilation phase. These are:

1) use of options (constant) where appropriate,

2) reprogramming to avoid type 6 links to non-existent segments, and

3) reprogramming to use the new system primitives (hardcore and user-ring).

It is probably better to only do the first of these as part of the initial recompilation.

Entry Meters

As mentioned earlier in the discussion of PLDT´s, the capability of metering selected entry points of the prelinked set of programs is provided at a very slight overhead for those entries metered. The method used is to replace the snapped links to those entries to be metered with pointers to a ring 1 gate procedure which counts the calls by updating a ring 1 data base (contained in the gate itself for efficiency). The dynamic linker can do the same by recognizing that a snapped link represents an entry in the list of entrypoints being metered. The actual overhead for the meter is 6 instructions, all executed in ring 1. There is no overhead for entrypoints not being metered. The 6 instructions will very rarely cause any page faults as all data referenced will likely be in core anyway at the time.

Note that this type of metering is possible because all processes will reference the entrypoint by the same virtual address and that it is therefore possible for the ring 1 metering program to know who to pass control to after updating the meter.

Appendix I

Preliminary Installation Plan

1. Install a system which places names in the AST name table.

2. Install a system which uses the AST UID hash scheme. This implies a reformatted AST. The DTBM will effectively be placed in the AST.

3. Install a system using the new KST/RNT strategy. This means that the reference name management will be completely split apart from the KST. Pathname resolution will be done by find_ which may keep a small associative memory for pathnames. The RNT and its manager will still remain in ring 0.

4. Install the first prelinking system. It will not require copy-on-write nor will it know about or handle new format object segments.

5. Install object_info_, trace, and the debuggers which recognize and handle new format object segments. No new object segments will appear yet.

6. Install the second (final) prelinking system. This system will handle copy-on-write, the isot management, and new object segments.

7. Install user-ring primitives for name space management. These will eventually go into ring 0 (one by one if need be). Some of these primitives were installed in step 3 above.

8. Install the new binder which accepts and generates new object segments.

9. Install the new PL/I compiler and ALM assembler.

10. Primitive swap. Several user-ring primitives will be moved into ring 0, and several hardcore primitives will be moved out of ring 0 (including hcs_).

11. Remove the linker and the RNT and its manager from ring 0.

12. Recompile the world. Reprogram where necessary to use new primitives, minimize static and obey new conventions.