TO: MTB Distribution

FROM: Richard Bratt

DATE: 10/28/75

SUBJECT: Prelinking

Introduction

This MTB describes the current design and implementation of a segment prelinker for Multics. It is assumed that the reader is familar with the MTB written by Steve Webber in which the initial design for a Multics prelinker was described, MTB-169. The design presented in this MTB, which represents an evolution of Steve's earlier design, attempts to capture the performance improvements inherent in prelinking while avoiding some of the problems implicit in the original design.

Background

Multics owes much of its elegance and flexibility to the policy of delaying bindings until the last possible instant. The Multics dynamic linking mechanism provides an excellent example of the beauty of delayed binding. In principle, the Multics dynamic linker allows each user to replace any (nonhardcore) system module he wishes with his own private version of that module. This ability to easily alter the environment perceived by programs even allows users to replace hcs_ by a user ring transfer vector, giving the user the ability to manipulate the hardcore interface seen by a program! An additional beauty of resolving all links dynamically is that a process will only bring into its address space those segments which it needs. Unfortunately, elegance and flexibility are not without cost.

Dynamic resolution of all links would imply that each process which used the PL/I compiler would have to pay the cost of resolving inter-module links within the compiler. Additionally, severe naming problems would arise from the global scope of the reference name space used by the Multics dynamic linker. The user of a Multics system which supported only dynamic linking would have to be very careful not to name one of his programs in conflict with the name of a module of the compiler unless his intent was to replace that compiler module. As a result, the normal user would likely view the flexibility of being able to supply his own private code generator as an expensive annoyance at best. The repetitive task of snapping inter-module links within the PL/I compiler is but one example of the negative performance effects of total reliance upon dynamic linking. Since each module is contained in a segment, and modules in a well structured system are small, we must expect extensive page breakage and ASTE contention.

Multics uses a static binder to link together major subsystems both to reduce the cost which would result from resolving all inter-module links dynamically and to limit the system's impact upon the reference name space of the linker. The static binding of modules contributes to improved system performance in several obvious ways. First, links snapped statically need not be resolved dynamically by every process. Second, page breakage is reduced. Third, fewer segments need be known to a given process. Fourth, a process' reference name space is smaller. Less obvious benificial performance effects include the reduction of the system's global name space (symbolic name to object mapping).

Static binding, while it has many advantages, is far from a panacea. Assuming object map machinery like that present in Multics, static binding can have negative performance effects. Imagine that each major language translator and editor were bound in different 256K segments. Since very few 256K ASTEs exist (on the order of 10 at MIT), we would expect severe AST thrashing in the pool of 256K ASTEs. Static binding may also force an increase in the amount of primary memory required to support the system's object map. Static binding can also increase the working set of a Multics process by causing related, but unused, information to be brought into a process' address space. This effect is most apparent in the combined linkage segments of a process.

Performance issues aside, static binding does reduce system elegance and flexibility in that reverting a statically snapped link requires both knowledge that the link was snapped, and the creation of a private bound segment with only one component replaced. The need to replace an entire bound segment to change a single component of that bound segment places an additional penalty upon the user replacing system code since he is forced to unshare the pure code of the other procedures in the bound segment. (1) The effect is also felt by other users due to the increase in the system's working set.

Basic Design

In MTB-169, Steve proposed an extension to the static binder. His goal was to create the capability to resolve all links in the system libraries once per bootload. Since in excess of 90% of all linkage faults are between system modules, this capability might allow us to increase system performance by nearly six percent, the fraction of system resources spent in dynamic linking.

Hardware restrictions that prevent the binder from creating multi-segment object segments (not to mention the strain that would be placed upon the system's object map machinery) force the prelinker to assign segment numbers at prelink time. As a result, the prelinker must overcome both the disadvantages inherent in static binding of name to object as well as the disadvantages inherent in the premature binding of segment

---

(1) It should be noted that the inability to unsnap links resolved by the binder is only an artifact of the current Multics binder. System flexibility could be enhanced (at the expense of performance) by modifying the binder to generate links which can be unsnapped.

numbers to object. (1)

The essential functional capabilities of a prelinker, culled from the numerous enhancements included in the original proposal, are summarized below. A prelinker must be given a specification of which modules to prelink and a specification of how to resolve name conflicts within the prelinked set. The linkage of these specified segments must be combined into combined linkage segments and a linkage offset table must be built by the prelinker to aid the prelinked process in locating its linkage. This requires that segments to be referenced in the prelinked process be assigned segment numbers by the prelinker. Finally the prelinker must scan these combined linkage segments and, using a set of search rules provided, attempt to snap every link encountered. (2)

To create a prelinked process the system must establish, in a virgin process, the correspondence between segment numbers and objects generated by the prelinker in a virgin process. At ring initialization time, the system must then copy the combined linkage segments and the linkage offset table from the templates generated by the prelinker if the given ring was prelinked. The prelinked process is then runnable. When a prelinked process gains consciousness it looks as if it had been running for some time, had snapped many links, and then had reinitialized its stacks and reference name table.

At this point we will explore the properties of the prelinker outlined above. It should be obvious that the prelinker cannot, in general, combine linkage containing a first reference trap. Similarly, it cannot snap trap-before-link links. Two less obvious classes of links unsnappable by the prelinker also exist. The prelinker cannot snap links to copy-on-write segments since the copy-on-write mechanism is ill prepared to handle multi-ring objects. (3) The current Multics design

---

(1) One of the advantages prelinking has over the current Multics binder is that links it snaps may be unsnapped.

(2) Naturally the prelinker must snap links relative to the segment numbers it assigned for use in the prelinked process.

(3) Copy-on-write is a total loss for a segment used by more than one ring since it assumes the ability of one ring to unbind a segment number from an object without regard to the validity of that segment number to object binding in other (possibly lower numbered) rings. Fortunately for Multics, the address space manager is not so naive as to allow one ring to unbind a segment number which is being used by a lower numbered ring. As a result, the copy-on-write mechanism is not guaranteed to work on multi-ring segments!

assumption that the linkage of a procedure has been combined before the procedure is invoked also prevents the prelinker from snapping links into the text of a segment whose linkage is uncombined. (1)

A further complication arises when we consider the ring of execution of the prelinker. It would be a clear mistake to allow the prelinker to run in ring zero. However, if the prelinker runs in some outer ring, then it cannot search the definitions of lower ring gates. (2) Adopting a suggestion by Steve Webber, the prelinker approaches the problem of searching inner ring definitions by making gates actors which, when called at location zero with a name, search their own definitions and return the offset corresponding to that name. (3) One very attractive feature of this design is that the hardware-validated ability to call the gate validates the caller's right to search the definitions.

MTB-169 contains one serious oversight which should be corrected at this point. The combined linkage segments (combined static segments, stacks, etc.) produced by the prelinker must be per-ring, not 1, 7, 7 as MTB-169 states. A ring must not be allowed to modify another ring's linkage! With this exception and the addition of "gate actors" mentioned earlier, the prelinker described above corresponds to the core of the prelinker described in MTB-169. We will now consider each prelinker enhancement proposed in MTB-169 as we complete the description of the current prelinker design.

Frills

The original prelinker design allowed reference names to be added to a segment in the pldt. This practice adds considerable obscurity and descriptive complexity to the system.

---

(1) This requires that the prelinker prelink rings in order of increasing ring number.

(2) Needless to say the prelinker must not be allowed to prelink a lower ring than the ring in which it is executing.

(3) For now just hardcore gates will be so constructed. When the user ring linker comes along the mechanism will be extended to include all gates.

(4) Anyone familar with hardcore should attempt to find the source for the procedure terminate. It cannot be found by the name terminate! It must be known by the seeker that the hardcore header, generate_mst's analogue of the prelinker's pldt, adds the name terminate to the segment makeknown. The information does not exist in the Multics naming hierarchy.

(4) The claimed advantage of saving directory space appears to be outweighed by the decentralization of the segment name space, the difficulty of describing the search rules used by the prelinker, and the fact that with identical search rules the prelinker and the dynamic linker can resolve a link differently. (1) As a result, the "refname:" pldt statement described in MTB-169 was not implemented.

The next feature of MTB-169 with which I wish to deal is the creation of a user reference name table (URNT) and a system reference name table (SRNT) by the prelinker. The physical fragmentation of a process' reference name table complicates reference name management and is aesthetically unpleasing. However, despite this drawback, it seemed possible that a significant performance gain lay hidden somewhere within the idea of a system reference name table. This hypothesis was experimentally tested. It flopped. The system reference name table effected only a factor of four data compression over the prelinked directories and, since these directories had to be touched often for other reasons, the SRNT increased the system working set substantially. As a result, its paging behavior actually degraded system performance. The SRNT is dead, long live the SRNT. (2)

The next feature retained in the current prelinker, is the calculation of access and the creation of a KST at prelink time. If a segment in the prelinked process' address space has a single access control list term of "*.*.*", then access is preset in a template KST and DSEG and the date-time-branch-modified is set in the template KST to allow validation of the preset access at AST connection time. At process creation time the template KST and DSEG generated by the prelinker are copied into the prelinked process' initial KST and DSEG and KST initialization is bypassed.

---

(1) Unfortunately, allowing a subset of a directory to be prelinked also complicates the description of the prelinker since it allows the prelinker and the dynamic linker to resolve links differently given the same search rules. This could be avoided by making the prelinker refuse to snap a link if it finds the target of the link has not been specified in the prelinked set but resides in the directory currently being searched. In the initial release of the prelinker search rules will apply to the subset of the directories involved as defined in the pldt. If this inelegancy results in too much confusion the strategy outlined above may be substutited in a future release.

(2) It remains an open question whether the SRNT is a viable concept on a system with enough memory to absorb the increase in system working set without causing excessive degradation in the system's mean headway between page fault. I suspect that the SRNT would best serve Multics by remaining buried.

The KST and DSEG of a process are ring zero segments which cannot securely be created by the prelinker running in an outer ring. As a result, a new gate will be added to hardcore which, given a specification of (object, segment number) pairs, creates a ring zero KST and DSEG for the prelinked process. Similarly, a new gate will be added to hardcore which allows a process to destroy a template address space if the process has modify permission on the containing directory.

The last feature proposed in MTB-169 which I wish to address deals with the prelinker "post pass" in which random stack header pointers, iocb pointers, and so forth would be filled in. It seems inadvisable to let the prelinker learn more about and imitate more of Multics initialization then is already the case. As a result, the initial release of the prelinker will not have any such post pass. This feature may be included in a future release after more careful study.

Administrative Control

The creation of prelinked processes is controlled by the -subsystem (-ss) login argument which, in conjunction with the principal's PDT entry, allows the pathname of a directory containing a prelinked subsystem to be specified. If the target subsystem does not exist, is found to be inconsistent, or was not regenerated within this bootload then no process is created. (1) Otherwise, the system establishes the template address space located in the specified directory in the new process and forces the process out of the warmth and comfort of ring zero into the cold hard reality of outer ring existence.

Outstanding Problems

The need to prelink subsystems every bootload presents a severe administrative problem. If we allow users to prelink private subsystems then the first time a user wishes to login to his prelinked subsystem within a bootload he must know to first login a standard process, prelink his private subsystem and then logout and log back in to his prelinked subsystem. Thereafter he need only login to his subsystem. But how can a user know whether he must prelink?

---

(1) Numerous reasons exist for the restriction that a subsystem must be prelinked within the bootload in which it is used. For example. A different hardcore system might be booted with a higher hardcore segment count. This would invalidate a subsystem prelinked in an earlier bootload. The recreation of >system_library_1 at bootload time also invalidates a template address space built in a previous incarnation of a Multics since the unique identifier of >system_library_1 changes.

In addition to this administrative burden on the user, allowing a user to prelink his own private subsystem may well degrade overall system performance by increasing system working set. If, however, prelinked subsystems are shared among users then paging improvements due to the sharing of combined linkage segments across processes may actually decrease the system's working set, increasing system performance. Prelinking thus seems to represent yet another instance of the age old "problem of the commons". It is to the advantage of any given user to run a prelinked subsystem tailored to his particular needs. However, if all users persue this course of action system performance will probably suffer.

For "the good of all" it would be advantageous to have relatively few prelinked subsystems shared by many users. However, in a scenario in which many users share a prelinked subsystem, the problem a given user faces in deciding when a prelinked subsystem must be prelinked is severe. Rather than attack this problem, I propose that initially only a small number of system supplied prelinked subsystems (maybe only one) be available for use. The responsibility for reprelinking these subsystems each bootload can then be shouldered by the system.

A site providing prelinked subsystems must decide between two initialization schemes. It may prelink the subsystems it supports through calls to the prelinker embedded in its answering service startup exec_com or it may spawn a process, Prelinker.SysDaemon, to prelink these subsystems. The former scheme has the disadvantage of effectively increasing bootload time. (1) The latter scheme has the disadvantage of introducing an indeterministic interval (hopefully a small one) between the time the system comes up and the time when users of prelinked subsystems may successfully login.

One final problem should be mentioned. Prelinked processes have a "chicken and egg" problem. This problem derives from two facts. First, prelinked processes use the copy-on-write mechanism to delay copying the template linkage segments created by the prelinker until they actually write into them. This, theoretically, will increase sharing when the use of separate static becomes widespread. (2) Second, the copy-on-write

---

(1) The increase in bootload time due to the time to prelink several subsystems may be substantial. In my experimental run prelinking >sl1 and >sss in ring four took about one half of a minute of virtual cpu time.

(2) It is not an absolute certainty that the presumed sharing will amount to a significant savings when weighed against the extensive working set of a Multic system (remember, only eligible processes can effectively compete for primary memory!) and the

mechanism currently executes in the ring of the "no write permission" fault. As a result the copy-on-write procedure and the transitive closure of the modules it calls must be prepared to operate while their linkage (and static) is being copy-on-written or, worse still, before their linkage can be combined! (1) This requirement does not obtain in the current system. Therefore, until a more suitable solution is agreed upon, a combined linkage region must be allocated in the initial stack of a prelinked ring (accomplished with a "linkage: stack,n;" statement). This linkage region, which is copy-on-written by the ring zero copy-on-write handler at makestack time, must either have the entire copy-on-write mechanism prelinked into it (prelinking all of >sl1 is more than sufficient) or have room at the end of the stack linkage region into which the linkage of the copy-on-write mechanism can be combined.

Three more permanent solutions to the prelinked process bootstraping problem under consideration follow. One, sharing of combined linkage sections by prelinked processes could be abandoned. This has the obvious disadvantage if sharing of linkage is found to be viable as well as the disadvantage of making copies of linkage segments which might never be referenced. Two, the copy-on-write mechanism could be made a system fault handler, executing in ring zero where its linkage is combined by the hardcore prelinker. This has the disadvantage of removing the copy-on-write mechanism from user control. Three, the copy-on-write mechanism could be made robust enough to copy its own linkage and it could be required that the copy-on-write mechanism be prelinked.

Although I feel the design presented in this MTB is cleaner than the initial design, I still have some misgivings. (2) Our inability to predict performance of proposed system changes, has forced us to expend massive amounts of man power (person power?) to evaluate the merit of prelinking by performing a test implementation and measuring it.

---

potential increase in paging due to the non-locality of static and linkage cause by separate static. Separate static tends to increase a process' instantaneous working set but tends to reduce its overall working set. Until we accrue experience with separate static the relative importance of these two effects is not likely to be well understood.

(1) This would occur if the copy-on-write mechanism had not been prelinked and the linkage section into which its linkage was to be combined was an uncopied copy-on-write segment.

(2) A better approach to the problem may be the ability to suspend processes, decomitting their resources, and resume them at a later time.

Performance

To test the efficacy of prelinking an experiment was performed to compare virtual cpu time and paging rate of prelinked processes verses standard processes running a standard metering script (script .3 with five users). The prelinked set used for this experiment contained all of >system_library_1 and all of >system_library_standard prelinked in ring four. Ring one was not prelinked. (1) The experiment indicated a performance enhancement of about four percent in virtual cpu time and two percent in paging when comparing prelinked processes with standard processes. (2)

---

(1) An earlier experiment had shown, as expected, that prelinking all system libraries degraded performance significantly due to the predictable increase in the prelinked process' working set. For this reason, only the most heavily used of the libraries were prelinked.

(2) This performance improvement may be slightly less on service since it is believed that the metering scripts place undue emphasis upon dynamic linking.

Appendix I

Description of the Prelinker

The description given below is taken from the prelinker itself. The prelinker takes a single argument which is an absolute directory pathname. It locates the ascii segment "pldt" (prelinker driving table) in this directory, parses it, and uses the output of the parse to control its operation. As output the prelinker produces a listing segment containing information about its operation. It also produces a template KST, a template DSEG, and a collection of template stacks and shared and combined linkage segments in the specified directory. These segments contain the freeze-dried essence of an anonymous process. To reconstitute such a prelinked process all process creation must do is:

* make a new pdir for the process
* set up a PDS for the new process
* copy the KST generated by the prelinker into the pdir
* initialize the RNT which is normally allocated after the KST
* copy the DSEG generated by the prelinker
* breathe life into the process

At ring creation time the system must:

* determine if the target ring is prelinked: if not, normal ring creation is performed, otherwise
* initiate the appropriate stack created by the prelinker (it is copy_on_write)
* perform all ring creation functions except:
  lot initialization
  isot initialization
  linkage region initialization
  set lotp, isotp, sct_ptr, stack_endp, old_lotp in stack header

When the new process wakes up it will have many segments already known and many links already snapped. It will, however, have no memory of how these links were snapped i.e. it will have a virgin reference name table.

The operation of the prelinker is controlled by the contents of the pldt as specified below:

SYNTAX:

<pldt> ::= <header> <body> <end stmt>
<header> ::= <max segno stmt> <search rule stmt> | <search rule stmt> <max segno stmt>
<body> ::= <ring group> [<ring group>]
<ring group> ::= <ring stmt> <ring body> <end stmt>
<max segno stmt> ::= Max_segno: <integer>;

```
<search rule stmt> ::= Search_rules: <rules>;
<rules> ::= <pathname>, <rules> ¦ referencing_dir [,  <pathname>]
<ring stmt> ::= ring: <ring> [, <ring>];
<ring> ::= <integer>
<end stmt> ::= end;
<ring body> ::= <ring body stmt> [, <ring body stmt>]
<ring body stmt> ::= <linkage stmt> ¦ <object directive>
<object directive> ::= directory: <pathname>, -all; ¦  directory:
          <pathname>; <segment stmt> [<segment stmt>]
<segment stmt> ::= segment: <name>;
<linkage stmt> ::= linkage: <class>, <size>;
<class> ::= stack ¦ shared ¦ combined
<size> ::= <integer>
```

In addition to the syntax rules given above, each ring statement must be followed by exactly one instance of each class of linkage statement before the first directory statement is encountered.

PLDT EXAMPLE:

```
Max_segno:          512;
Search_rules:       referencing_dir,
                    >system_library_1,
                    >system_library_standard,
                    >system_library_unbundled;
ring:               1;
directory:          >system_library_1, -all;
directory:          >system_library_standard;
     segment:       bound_mseg_;
end;
ring:               4,5;
directory:          >system_library_1, -all;
directory:          >system_library_standard, -all;
directory:          >system_library_unbundled;
     segment:       bound_fortran_;
end;
end;
```

SEMANTICS:

* The Max_segno statement specifies the size of the lot and isot in the prelinked process.
* The Search_rules statement specifies the search rules to be used during prelinking.
* The ring statement identifies a group of rings to be prelinked.
* The initial linkage statements in a ring group identify the size of the initial linkage region for the given class. Only the size of the stack linkage may be zero in which case no combined linkage region is allocated in the stack. Linkage of class "shared" is allocated in sl_x.yy and linkage of class "combined" is allocated in the stack linkage region (if one exists) and in cl_x.yy. When a linkage region of a given class

is exhausted a new linkage region of the same type is allocated (except in the case of stack linkage in which case a new combined linkage region ,cl_x.00, is allocated).

* A directory statement with the -all option specifies that all segments in the named directory (and segments linked to from the directory) be prelinked in the rings specified by the containing ring group.
* A segment statement explicitly names a segment (or link) within the previously named directory which is to be prelinked.
* Subsequent linkage statements within a ring group redefine the size of linkage regions of the specified class and force a new linkage region of the specified class to be allocated. N.B.: Any subsequent linkage statements of class "stack" force subsequent linkage out of the stack and into cl_x.00.

OPERATION:

```
parse the pldt
process rings 1 to 7 in order
for each ring:
     for each segment to be prelinked:
          initiate segment
          remember its names as reference names
          combine linkage of segment unless:
               not object
               not in execute bracket
               copy switch set
               trap at first reference
     for each linkage region combined:
          snap all links unless:
               target not in prelinked set
               trap before link
               definition not found
               definition into uncombined region
 call hardcore to construct a template address space
```

(exception: separate static is not initially combined. If the prelinker attempts to snap a link into uncombined static whose linkage is combined, then the prelinker combines the static and snaps the link)

RANDOM FACTS AND ASSERTIONS:

* To simplify initialization of prelinked processes, the stacks and linkage segments produced by the prelinker will be copy_on_write segments known in the prelinked process. Only the template KST and DSEG must be explicitly copied by process creation.
* All segments known to the prelinked process which had a single acl term of *.*.* at prelink time have access preset in the DSEG and KST. This does not constitute a violation of security since the dtbm stored in the KST when access was checked by the prelinker will be used to validate the saved access before the

SDW in question is connected.
* Isot slots which are brothers of lot slots set by the prelinker are either set or marked to cause an isot fault when loaded by a load packed pointer instruction and have the offset of the virgin static in the lower 18 bits.
* The lot and isot created by the prelinker reside in the stack.

Appendix II

Description of the Template Address Space Manager

The description of the template address space manager, like the description of the prelinker given above, is extracted from the appropriate source modules. The reader is asked to bear this fact in mind while reviewing this section. template_address_space provides functions for creating and deleting a template address space. A template address space is stored in a directory and is defined by two ring zero segments named "template_kst" and "template_dseg".

---> call template_address_space$create (dirname, access_calculated, code)

create makes the segment "kst_seg" in the given directory known. It assumes that this segment has the format of a kst. create makes the following demands of its environment and the given segment:

* kst.lowseg = active_all_rings_data$stack_base_segno
* kst.highseg < active_all_rings_data$max_segno
* kst.highest_used_segno > kst.lowseg+7
* kst.time_of_bootload = sys_info$time_of_bootload
* kst.highest_used_segno is consistent with bitcount
* caller has read access to kst_seg
* each kst entry from lowseg to highest_used_segno contains:
    * a valid uid or a uid of "0"b
    * baseno (kste.entryp) is valid w.r.t. this kst
    * usage counts
* the caller's address space contains every object in the kst to be built
* no segment named "template_dseg" or "template_kst" exists in the given directory

create makes two ring zero segments named "template_kst" and "template_dseg" with r *.*.* access. create then uses the information in kst_seg and its callers' address space to transform kst_seg into a secure, valid address space template. If an object in the new address space has a single acl term of *.*.* then access is precalculated in the template address space. If any inconsistencies are found in kst_seg or if any of the assertions above are found to be violated then "template_kst" and "template_dseg" are deleted and an error is returned. If the address space template is successfully built then kst.template is set to help template_address_space$delete validate its right to delete this template address space and the count of segments which had access precalculated is returned.

---> call template_address_space$delete (dirname,code)

delete is called to delete the template address space (template_kst, template_dseg) stored in a given directory. delete requires that the given directory contain two segments named "template_kst" and "template_dseg". These segments must have ring brackets of 0, 0, 0 and the caller must have modify permission to the containing directory. delete validates that the kst is marked as a template kst. Unfortunately, since we don't have property lists, delete cannot be absolutely certain that it is deleting a (template_kst, template_dseg) pair created by template_address_space$create. We assume, somewhat nervously, that the checks made by delete are sufficiently safe to prevent users from destroying ring zero segments not created by template_address_space$create.

Appendix III

Enhancements to the Prelinker

At this time several possible future enhancements to the prelinker outlined in this MTB have been identified. These enhancements deal exclusively with the pldt accepted by the prelinker. One possible enhancement would allow the use of star names in "segment:" statements. The other enhancement currently under consideration is the addition of an "exclude:" statement to the pldt which would allow certain segments identified by "segment:" statements to be exculuded form the prelinked set.