PROJECT MAC

Computer Systems Research Division

May 22, 1973

Request for Comments No.23

PROPOSAL TO REMOVE MULTICS DYNAMIC LINKER FROM THE SECURITY KERNAL

by Philippe A. Janson

Abstract: The motivations for removing the procedures and data bases directly related to the dynamic linking mechanism is first outlined. A brief description of the present linker is then given. Finally,the major problems resulting from the new design are explained.Tentative solutions are given for each of them. Three major design changes are proposed: a hardware modification in the association between access modes and ring brackets, a new prelinking mechanism for the linker, a modification in the dynamic linking mechanism itself.

(Note: I will not be here during the summer. Comments on the paper are welcome. They can be dropped on my desk in room 517. Thank you.)

This note is an informal working paper of the Project MAC Computer Systems Research Division. It should not be reproduced without the auther's permission, and it should not be referenced in other publications.

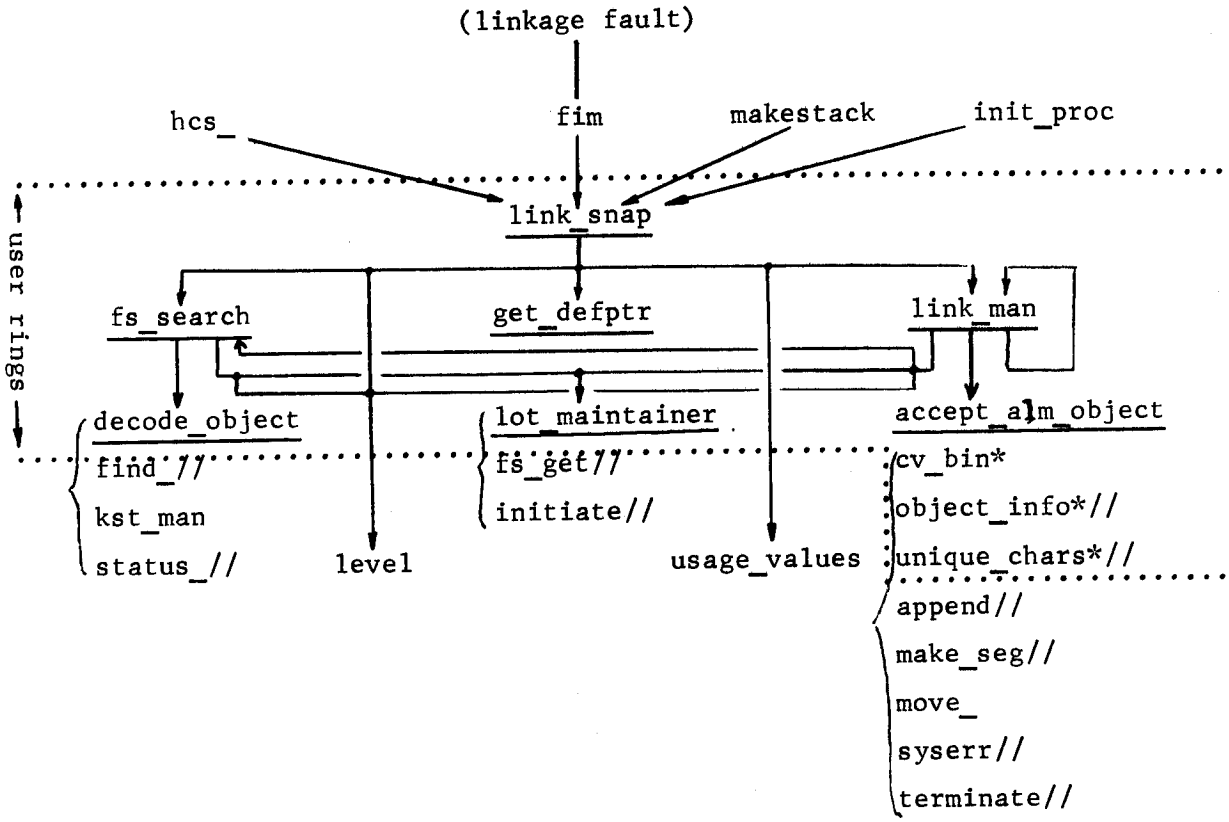
I. Motivations.

The idea of removing the dynamic linker from the most protected area of Multics, namely ring 0, can briefly be stated as follows. There is apparently no reason why the linker should need more privileges than the faulting procedure on behalf of which it is acting. This is obvious in the case where the number of the target ring of the link being snapped is greater than or equal to the number of the faulting ring. As will be seen in this proposal, the linker needs no more privilege than the faulting ring even in the case that the target ring has a lower number. Thus the principle of least privilege suggests that belongs in the faulting ring. Not only is it desirable to allow the linker to execute in the faulting ring, but it is desirable to force it to do so. As will become clear, the notion of dynamic linking is almost irrelevant to ring 0 because ring 0 is almost totally prelinked. Preventing execution of the linker in ring 0 would avoid the necessity to audit it to certify the system. This paper will explain how the linker can be removed from ring 0. We do not claim that the solution proposed are the best ones, nor do we claim that they are all easy to implement. We just hope so and would be glad to hear any comment on this point.

II. Brief description of the dynamic linker.

The dynamic linking mechanism is supported by a set of procedures known as the linker. The linker can be entered on a call, to force a link to be snapped, or more generally, is invoked on a linkage fault, i.e. the attempt to reference a location unknown by the running process at that time. Snapping a link is done in the same way whether the linker is called explicitly or entered on a fault. We will show in this paragraph what procedures are involved in the process of dynamic linking, and we will define wich among these are proper parts of the linker and should be removed from ring 0. These definitions will become clear and be justified in subsequent paragraphs.

1) Trace of the calls in the linker.



Underlined procedures are part of the proposed bound linker and will execute in the ring where a link is to be snapped.

* indicates procedures called by the bound linker which will execute in the ring from which they are called.

Other procedures will execute in ring 0 as parts of the hardcore.

// indicates that the tracing was stopped (uninteresting).

2) Description of the linker.

Assuming that the reader is familiar with the general concepts of dynamic linking, we will briefly define what each of the procedures of the linker does and explain why other procedures are not -properly speaking- parts of the linker. We will also explain why these shouldn't or couldn't be removed from ring 0. Problems and solutions involved in removing the linker from ring 0 will be outlined further.

a/ link_snap.

This is the main procedure of the linker. All calls to the linker are directed to it. It takes a pointer to the machine conditions, or to the link to be snapped, or a segment name and an external symbol name as input. It returns to the caller when the link is snapped. To snap the link, it analyses its type (see MPM for the definition of types), and performs calls to fs_search, get_defptr and link_man accordingly.

b/ fs_search.

It is invoked by link_snap every time the linker needs to snap a link to a segment referenced by its name. Fs_search picks up the search rules from the kst and searches all subsequent directories of the rules until it finds the name of the segment. It eventually initiates the segment, makes it known and then returns its segment numberto the caller.

c/ get_defptr.

It is invoked by the linker when it needs to find the offset of an external symbol definition in the definition section of the external segment.

d/ link_man.

This is probably the second most important part of the linker after link_snap. It performs all the necessary bookkeeping to manage the LOT's, the CLS's and the combined linkage sections. The present linker implements a rule according to which the linkage section of an object segment has to be copied into a virgin segment (for gates) or into a combined linkage segment when the object is first referenced. Link_snap calls link_man for this purpose. Link_man checks to see if a linkage section exists for the target segment of the link being snapped. If it is not the case, it creates one. In order to do this, it checks to see if a LOT exists in the target ring. If not, it creates one. When the LOT exists, an entry is created in it for the linkage section of the target segment. All above checks involve calls to the lot_maintainer. Recursive calls to link_man result from a search in different rings (origin and

target rings) to find a possibly existing linkage section. In the future design, link_man will be totally separated from link_snap. The "rule of the first reference" will no more be respected.

e/ decode_object and accept_alm_object.

These are utilityroutines used by the linker to get information about object segments.

f/ Other procedures.

We will very briefly describe here the procedures involved in linking but not parts of the linker and we will explain why they must execute in ring 0.

-level is a short utility routine used to set or get the validation level. This level being a protected item, the routine should obviously be kept in ring 0 and be accessible through a gate. Moreover before accepting a request,the routine should make sure that the caller has the right to set the level it wants.

-usage_values is a metering routine. If counters have to be trusted by the system,they obviously should be in ring 0. Therefore the routines updating them should also be in ring 0. (In fact the future linker will not even access the routine, because it is unsafe to allow the users to update counters).

-cv_bin, object_info and unique_chars are also utility routines. They can eventually run in user rings but may also have to run in ring 0because many procedures currently running in ring 0 need them and outward calls with inward returns are forbidden.

-syserr is a short procedure called of system error. It needs access to many protected items and since it deals with system errors and not user errors it should obviously be protected from the user rings, i.e. run in ring 0.

-terminate (makeunknown), move_, makeseg, append, initiate, fs_get, status_ and find_ are all file system procedures: they deal with directories segments, links and branches. The idea of removing the whole or a part of the directory control out of ring 0 has been around for some time. We have no measure of the feasibility of the project. All we know is it is beyond the scope of this project. For the time being we will consider that the directory control has to stay in ring 0,if not for sound reasons at least for practical reasons.

-kst_man has to stay in ring 0 because it manages the kst which is a protected data base (because it may contain information about all rings).
-lot_maintainer may execute in ring 0 because it is part of the prelinker. But when called by the linker it should execute in the faulting ring. Actually, the present lot_maintainer will be used only by the prelinker and a copy of it will be made part of link_man.

g/ Comments.

In the above text, the reader will have understood that by removing from ring 0 we mean setting the ring brackets to (1,5,5). By allowing execution in all rings we mean setting the ring brackets to (0,5,5) and by forcing execution in ring 0 we mean setting the ring brackets to (0,0,0).

Some other procedures not mentionned above can be removed from ring 0 because they are directly related to the dynamic linking mechanism although they are not part of the linker. Unsnap_service, and initiate_search_rules are in that category.

III. Problems involved in removing the linker from ring 0.

1) Goals.

We see the future linker as a set of procedures bound together in one segment with ring brackets (1,5,5). When a process starts running in any user ring, the linker segment will be prelinked to the rest of the system by some mechanism to be defined. Later on, as the process takes a linkage fault, the fault interceptor module will somehow call the linker in the faulting ring to snap thelink. As this is done, the linker will restore the machine conditions and execution will proceed.

2) The problems.

In the coming paragraphs, we will explain all the problems encountered in trying to achieve our goal. We will also outline tentative solutions for each type of problem. Although it is not always the case, we have tried as much as possible to justify our solutions in terms of simplicity, feasibility, and generality.

3) Problems of gates into the supervisor.

Removing procedures out of ring 0 has two direct implications on gates. First, one might hope that the procedures removed can still access (call)the ring 0 procedures through existing gates. Unfortunately, this turns out to be the case. For instance, since kst_man will be kept in ring 0, we must have gates to call it, e.g. hcs_$getkstep. Any such gate will have to be added to the existing gate segment. On the other hand, present gates will have to be removed. This is the second problem. For instance, hcs_$make_ptr will become obsolete since thelinker entry point link_snap$make_ptr will be outside of ring 0. This poses a bad problem of compatibility. All programs presently using such gates will have to be modified. One solution is to keep the present linker around for a while. All linkage faults would be directed to the new linker but explicit calls to the linker would be handled by the present linker for a transient period. Using the present linker could eventually cause messages to be directed to the output file to request the program to be changed before a given deadline after which the present linker would be removed totally. Another solution is to keep the obsolete gates for a while but to modify them somehow sothat they would redirect the calls to the linker in the outer ring.

4) Problems of arguments.

In creating the new gates for the linker, one might encounter arguments problems. If one created a gate like for instance hcs_$initiate_segcount(dirptr,refname,...), one would be in trouble. If this gate was called by the linker from ring 4, the hardware would refuse to access the directory pointed to by dirptr because directories are protected and cannot be accessed directly on behalf of a user ring. The solution is to use the pathname of the directory instead of the pointer. We have given here one example of argument problem. There exist many others. In all cases, the solution is to replace pointers to protected items by symolic information.

5) Problems of references.

The present linker needs to read several items in protected data bases. This will obviously be impossible in the future design. The data bases presently referenced by the linker are active_hardcore_data, pds and the kst. Ahd is a metering data base used to collect statistical information about the use of various ressources. We have already mentionned that the linker cannot access any ring 0 metering data base. Therefore, acces to ahd must be removed. If one wants to measure the activity of the linker, one can eventually create per ring datat bases out of ring 0. The kst is accessed to read the search rules. In this case, the search rules will be removed from the kst and be put probably in the CLS on a per ring basis (see further). The problem therefore disappears. Finally, the case of the pds can easily be resolved. Some of the information kept in pds will no more be necessary in the new design. Another set of items does not need to be protected and therefore can be put in per ring data bases (stack or CLS instead of pds). A third kind of items will be kept in ring 0 but is used only by make_cls_lot which anyway has to run in ring 0 as will be seen further.

6) Prelinking problem.

a/ System calls to the linker.

Before a process starts running in any user ring, the system presently makes three calls to the linker, all to link_snap$make_ptr. Makestack calls the linker to get a link to pl1_operators_$operators_table. This pointer has to be stored in the stack header. Another call, by makestack too, is to get a pointer to signal_$signal_for the same purpose. The last call to the linker is done by init_proc to get a pointer to the first procedure to be executed in the user login ring. Given the fact that there will be no linker in ring 0 in the future, and given that it is impossible to run the linker in a ring n (n greater than 0) to transfer tothe first procedure (itself!?!) in that ring or to make a stack for ring n, we have to find other ways of snapping these first links. One way would be to use the system prelinker but it is not desirable because it would force us to keep the prelinker around after system initialization. Instead, we can easily find the segment numbers of pl1_operators and signal_ by initiating them. Then, tofind the offsets of the appropriate entry points, for pl1_operators and signal_, we can use the corresponding offsets which are known in ring 0 since pl1_operators_ and signal_are

known in the SLT and prelinked in ring 0. The case of the initial procedure is a little more complex. Since we cannot use any linker in ring 0, the idea is to fabricate a conventional pointer to transfer to a short routine in the outer ring which calls the linker in that ring to snap a link to the actual first procedure. In other words, when a process is initialized, the last procedure of the ring 0 initializer transfers to init_proc$init_proc through a conventional pointer. Init_proc is a short alm segment, initiated in a kst template like the bound linker (see further), with ring brackets (1,5,5) like the bound linker. Control transfers to it by a simulated return (RTCD) from ring 0 to the appropriate outer ring. Init_proc first calls hcs_$get_initproc_name. This call causes a reference to the stack which in turn causes the linker to be prelinked (see further). As the result of this call, ring 0 returns the name of the first actual procedure to be executed for the process being initialized. Init_proc then calls the linker to snap a link to this first procedure and finally transfersto it through the pointer returned by the linker. Execution then proceeds normally.

b/ Prelinking the linker.

One can envision three different philosophies to prelink the linker. The first idea is to prelink it at the very last moment only when it is needed. By this we mean that it should have a combined linkage section with all links snapped in a ring just before it must be used for the first time in the ring. When it would be called for the first time on a linkage fault, the fim could test some bit in the ring stack header and call by a signalling mechanism, a prelinking procedure. This procedure would be a self-contained segment with no external symbolic references. It would have pointers to the bound linker and to the segments referenced by the linker. It would scan the bound linker linkage section, combine it and snap all the links. The problem with this design is a matter of economy. Although the design is theoretically feasible and clean, it would imply running the very same procedure with the very same data once for each ring where a process goes. Therefore, we will abandon here this first idea. The second proposal is quite naturally to have the linker prelinked only once for each process. This implies having a template for the snapped linkage section of the linker and copying it into the CLS of the appropriate ring when the process goes to that ring.

Since all rings have to trust this linkage section template, it has to be created in ring 0. This in turn implies that the prelinking procedure run in ring 0. Since we would like to avoid creating new procedures in ring 0, we would like to try to use the existing system prelinker to prelink the linker. On the other hand, we would like to avoid keeping the system prelinker after system initialization. This leads us directly to the third proposal: a system wide prelinking of the linker at system initialization time with the system prelinker. We strongly recommend this option because it clearly is the most economical and uses only existing procedures. We will now give more details about it.

The system prelinker has three major components: prelink_1, prelink_2 and the lot_maintainer. Prelink_lis of no interest to us here. It just scans the SLT to find all segments to be prelinked at system initialization time. Prelink_2, on the other hand, is what we are interested in. For each segment number passed to it, it snaps all the links to external symbols referenced by the segment. In our design, we add an entry point to prelink_2: the input arguments are a pointer to the bound linker (previously initiated in a system wide kst template), and pointers to the few segments referenced by the bound linker (hcs_, cv_bin, object_info and unique_chars previously initiated in the same kst template), and a pointer to a virgin segment,called linker.link (initiated with ring brackets (0,0,0) in the kst template). The modified prelink_2 then scans all links of the virgin linkage section of the bound linker and copies the snapped links into the linker.link segment. The linker.link segment would then act as a linkage section template: each time a process enters a virgin ring, the template is copied into the CLS of the ring.

At this point, the reader may wonder what all the implications of such a design are. The first one has to do with system initialization. At the present time, after the system is prelinked, the prelinker is discarded. In the future design, before the prelinker is discarded, the system initializer wuold have to initiate the bound linker, linker.link and segments referenced by the linker in the kst template and in the descriptor segment template. It then would call prelink_2 for the last time to prelink the linker. The second implication has to do with process initialization. When a system is initialized, the kst template will have to be copied into the process kst. This implies that the linker, the segments referenced by it and linker.link will have the same segment

numbers in all processes. This is not a restriction and poses no problem. If a user wants to write his own linker, it certainly is possible provided he copies a pointer to his linker in the stack header of his ring for the signaller (see further). The standard linker will always be initiated in all processes and can be used as a prelinker for any user provided linker.

8) Ring initialization.

By ring initialization, we mean the set of operations which have to take place when a process enters a virgin ring. Our first point is to show that the stack, the CLS, the CLS template for the linker and the LOT are all vital to a process running in a ring, and that they cannot be created from that ring. We want to prove that it is impossible for a process to create its own stack, CLS and LOT in a ring. The basic operation involved in all cases is the creation of a virgin segment. Since the directory control and the file system are in ring 0, the creation of a segment requires several calls to the supervisor. Calling from a virgin ring into the supervisor means a linkage fault and passing arguments means a stack push. Since we are precisely trying to fabricate a stack we cannot use one, and since we are trying to fabricate a CLS and to initialize it we cannot use it either, i.e. we cannot take linkage faults. Our second point is to prove that the stack will always be referenced before the CLS. The way a process references the CLS is by first finding a pointer to a linkage section in the LOT. To do this it has to read the pointer to the LOT which is stored in the stack header. No matter whether the process actually wants to reference the stack or the CLS first, it will always reference the stack first as explained above.

Given this principle, the following design has been set up for the ring initialization. On the 6180, stacks have reserved segment numbers. When a transfer is performed from one ring to another, the stack pointer is always set to the appropriate reserved number. If the ring is virgin, when the process first references the stack through the stack pointer, it takes a segment fault. As a result, the supervisor discovers that the stack is missing in the faulting ring and calls makestack. In our design, makestack calls make_cls_lot. This new procedure would create a CLS, copy the template linkage section for the linker in it, create the LOT, create a LOT entry for the linker linkage section in it, initialize all other LOTentries to a special packed pointer (see further). It then

would return a linker linkage section pointer and a LOT pointer to makestack. Makestack would store them in the stack header.

As a last point we will mention a few remarks about the search rules. They will be in the CLS on a per ring basis,in the linkage section of the linker as internal static variables. They are initialized at a default value (kst, parent directory, working directory). It is probably worth mentioning here that the form of the search rules will be different. Because fs_search will not be able to reference directories by pointers, we must represent the search rules by symbolic information (codes or pathnames).

9) Transfering out.

In the new design of the linker, each linkage fault will cause the fim to transfer control out to the faulting ring, to the linker. There exist several ways to do this. The best one seems to be the mechanism somehow similar to signalling but much simplified, and less expensive. When the fim handles a linkage fault it should call the signaller to copy the machine conditions in the faulting ring and to set up a stack frame for the linker. It then should call the linker through a pointer stored in a standard place in the stack header. After the link is snapped, the linker will return to the signaller. The signaller will reenter master mode and perform an RCU to restore the machine conditions. This mechanism seems to be the most appropriate one in this case. It has got all necessary features (setup stack frame, copy machine conditions, transfer out, return to master mode and RCU). It has no other expensive and unnecessary features. It already exists and takes only about a hundred machine instructions. This does not make a big difference on the total of the linker.

10) Hardware modification.

This paragraph explains one of the clever ideas suggested by Mike Schroeder. Let us consider the situation of linking to a gate into a lower ring. In order to snap the link, the linker must be able to find the entry point in the definition section of the segment. To do this, it has to be able to read the definition section of the gate segment, i.e. it has to have read access to the gate. One way to do this was to run get_defptr in the target ring, but this would have forced us to have a gate from each ring to each ring for the linker, which turns out to be unfeasible and ridiculous. Another easy but unelegant solution was to force get_defptr to execute in ring 0. The final solution, proposed by Mike Schroeder, is to associate the read access mode with the gate bracket

instead of associating it with the execute bracket. This would enable the linker to read gates from higher rings. The only disadvantage is that programmers must be aware of the fact that a gate of a protected subsystem is readable from higher rings. The hardware modification, according to Honeywell engineers is quite simple to perform.

11) The bound linker.

The only difference in the structure of the bound linker compared to the present linker is the absence of relation between link_man and the other procedures. Presently, link_man is called by link_snap to combine the linkage sections of target object segments on the first reference. In the new design this first reference rule is impossible. If the target segment is a gate, the linker , running in ring n, cannot combine the linkage section of the gate into a lower ring. The rule adopted is to initialize the LOT, as explained above, with special packed pointers, i.e. with pointers which when unpacked would cause a simulated fault. As a result of taking such a fault, the fim would call the signaller. Instead of calling the link_snap entry, the signaller would call the link_man entry. When returning from link_man, control would reenter the signaller, thereby resetting the master mode. The signaller would then do an RCU. The nice thing about this is that the linkage section of a segment is combined only at the very last moment when it is needed.

IV. Implementation.

The first phase of the work proposed is to draw exhaustive lists of all gate,argument and reference problems. The second phase seems to be the creation of the new bound linker segment and of necessary gates to simulate the new linker. This implies that all problems except prelinking must be solved at this stage. The final phase would be to modify the system initiaizer and to issue the new system to the users.