To: Distribution

From: Richard G. Bratt

Date: 12/12/74

Subject: A Proposal for Removing Name Space Management from Ring Zero.

Multics allows objects in its storage system hierarchy to be referenced by three distinct classes of names: path names, reference names, and segment numbers. The binding of these names to objects in the hierarchy is controlled by directory control, name space control, and address space control respectively. Currently the modules in the hardcore supervisor that implement these functions are more interconnected than need be. This MTB proposes a restructuring of address and name space control which allows name space control to be removed from the security kernel of Multics. Together with Phil Janssons previously completed user-ring linker , this design produces a simpler, smaller supervisor with a simpler interface.

Currently a process' name space has two distinct components: a segment name space and a directory name space. The segment name space associates names with non-directory segments. This name space is under explicit user control. That is, the process is free to associate any name or group of names with a segment. Furthermore, a process may dynamically modify its segment name space. The directory name space which associates names with directory segments, however, is not subject to explicit user control. Instead, it is managed by ring zero which constrains names of directories to be absolute pathnames of the directory.

The distinction between segment reference names and directory reference names seems somewhat artifical. A process should be free to associate any name it chooses with a directory. Consider how easily the working directory and search directory concepts fit into such a scheme. We could bind the name "working_dir" to a process' working directory and "search_dir_n" to its n'th search directory. A process could then reference these directories by name using the normal name space management mechanisms.

The primary goal of the design presented in this MTB is to remove name space management from the security kernel of

---

Multics. It has been argued that a serious consequence of any scheme which realizes this goal is that a process' name space can no longer reflect name changes in the hierarchy. This argument is based on a confusion between reference names and directory entry names. It seems obvious that a process does not want its name space to change without its consent. Changing a segment's name does not change a process' access to it. A prime advantage of reference names is precisely this ability to insulate a process from name changes in the hierarchy. We should distinguish reference names from directory entry names. A reference name is a name we temporarily bind to a segment. A directory entry name is a selector of a particular entry in a directory. We need directory entry names only to physically select a branch for the first time; after that we should be free to call it whatever we choose. If any valid reason exists for notifing a process that the names on a segment or directory that it is using have changed, the system could signal a name_change_on segment_x condition. This would require the addition of some sort of KST trailer mechanism to the system. This may eventually be necessary if for no other reason than Multics will eventually run for extremely long uninterupted stretches. If a process were to stay permanently logged in it would require notification of on-line installations. This in itself is a difficult problem which I do not intend to address here. The only point I wish to make is that the process and not the system should control the duration of name bindings.

While there does not appear to be any intrinsic need for the Multics security kernel to support name space management, its removal from ring zero is complicated by the fact that the current Multics address space manager, which provides a legitimate kernel function, depends on the name space manager. Specifically, the address space manager uses the name space manager to manage an associative memory of (directory pathname, segment number) pairs. It is therefore necessary to decouple address space management from name space management before the latter can be removed from ring zero.

The dependence of address space control on name space control manifests itself in the recursive procedure find_ which the address space manager uses to map directory pathnames into directory segment numbers. When find_ is invoked it calls the name space manager with the pathname it is given. If the name space manager returns a segment number then find_ is done. Otherwise, find_ splits the pathname into a pathname of the parent directory of the target directory and the name of the target directory. It then calls itself recursively to obtain a segment number for the parent directory. Using this segment number as a pointer to the parent directory, find_ attempts to initiate the target directory. If it suceeds it adds the pair (path name of target, segment number of target) to the name space manager's data base and returns.

This proposal suggests a radical change in the ring zero address space manager. The essential result of this change is that find_, as described above, need no longer be called by the address space manager. This allows both find_ and name space management to be removed from ring zero.

Currently, determining whether a process should be permitted to initiate an arbitrary directory is quite difficult since we wish to prevent a process from detecting whether or not a given directory exists unless it has access to that directory. This difficulty stems from the fact that the ACL of a branch and its physical storage map reside in its parent. Since we wish the ACL of a branch to exercise complete control over access to that branch, we must permit a process to initiate all superiors of accessible segments independent of access to these superiors! To avoid this difficulty, Multics inexorably couples the initiation of a directory with initiating an inferior segment. This inability to initiate directories directly has lead to many needlessly complex mechanisms for manipulating directories. In addition it has forced us always to refer to directories by pathname. Not only is this inefficient, but it requires that the address space manager be able to call find_. If we could initiate directories directly then we could use segment numbers as directory specifiers. Address space control could then take a segment number instead of taking a pathname as a directory specifier. Since address space control would no longer need to call find_ it could move out of ring zero along with name space management without compromising the security of address space control.

Actually, coupling directory and segment initiation does not solve the problem. Since a process cannot read the access control list of a segment until its parent is known, the system still must permit a process to initiate directories which it may not have the right to know exist! By causing the initiation of these superior directories to occur in a single, indivisable ring zero call, the system could, in principle, prevent security leaks. This could be accomplished by terminating those intermediate directories which had to be initiated only to find that the process had no access to the terminal segment, before returning to the caller. Unfortunately, the current system does not do so. This allows any process to determine the existence of any postulated directory. Certainly one approach is to correct this flaw in the current system. However, there seem to be many ways of forcing such a scheme to compromise information. For example, suppose a process filled up its address space intentionally and then called ring zero to initiate >secret>x. If ring zero was not very careful it might cause the process to die due to a KST overflow if and only if >secret existed. This would allow the existence of >secret to be inferred by whether or not the process died.

I propose that we decouple segment and directory

initiation. As was noted earlier the basic problem to be solved is how can the system decide whether a process should be allowed to initiate a given directory. There are essentially four schemes for making this decision. The first scheme involves recognizing that if the access control list of a directory is to completely express access to that directory we must make explicit the now "hidden" permission to initiate a directory if some descendent of the directory is accessible to the process. The obvious way to accomplish this is to invent a new directory access mode called "initiate". This mode allows the named principal to initiate a directory and to use the information it contains which is relevent to accessing descendents of that directory. This makes the decision of whether or not a process should be allowed to initiate a directory quite simple. If the process has non-null access to the directory then it may initiate it. Otherwise, it may not. Unfortunately, this scheme defeats our desire to have the access control list of a segment or directory completely express what processes may access that segment or directory.

A second way to decide whether a process may initiate a directory is to search the hierarchy subtree rooted at that directory. If the process has non-null access to any member of this subtree then the process should be allowed to initiate the directory in question. Naturally, this scheme is far too inefficient to consider seriously.

A third method of deciding whether a process may initiate a directory is to require non-null access to the directory. This scheme has the disadvantage, shared by the first scheme discussed, of preventing the access control list of a directory or segment from being the sole arbiter of access to that directory or segment. Inorder to initiate a segment a process would need non-null access to the superiors of that segment.

I propose that we take a forth approach to the problem of initiating directories. Instead of worrying about whether or not a process has the right to initiate a directory let us allow all processes to initiate any directory - whether or not it exists! The key to this scheme is preventing the user from detecting any difference between an initiated directory which does not exist and an initiated directory which exists but which the user has not proven his right to know exists. How this is to be done will be discussed later. The ring zero address space manager interface resulting from this approach seems quite natural. Ring zero no longer concerns itself with pathnames. Instead, it accepts directory segment numbers for directory specifiers. To allow this scheme to bootstrap itself we will define the segment number of the parent of the root to be zero. Initiation of segments and directories will be controlled by initiate_ which will accept a parameter specifing whether a segment or directory is to be initited. The rationale behind

distinguishing directory and segment initiation is that a process usually has a preconceived idea about the type of a branch it wishes to initiate. When reality does not support this preconceived idea the process is usually in error. Forcing the process to make explicit the type of branch it is expecting allows ring zero to immediately catch all such errors. This prevents a careless process from bumbling along thinking all is well only to die when it attempts to access a directory as a segment or vice versa.

An important consequence of not handling pathnames in ring zero is that file system links can no longer be interpreted in ring zero. This requires that links be readable in the outer rings which raises the question of what, if any, access control should be placed on reading links. The simple approach, which is taken in the current system, is to make links completely public, readable in all rings by all processes. This has the disadvantage that if some process can guess the pathname of a real link then it can prove the existence of the parent directories of that link. At the other end of the spectrum we could place access control lists on links thereby explicitly naming those processes which may read the link. This seems a bit too bulky. I propose that we consider a link to be part of its containing directory, readable only by processes having status permission on that directory. This scheme has the virtues of being simple, easy to implement, and plugging the information hole which uncontrolled access to links provides in the current system. While this scheme does make one class of currently legal uses of links illegal, this restriction does not seem too severe.

When initiate_ encounters a link it will return the link and a status code which informs the outer ring procedure that a link was encountered. The outer ring procedure may then try the new path specified by the link. Since this is happening in an outer ring we need no longer have a standard interpretation of links. That is unless the function moves out of the kernel but not out of the supervisor. If ,however, it resides in the user ring the process may interpret links in any manner it chooses. Why not let links contain relative pathnames ,offsets, or even arbitrary character strings? The important point is that while the kernel may be the keeper of links it does not interpret them. Naturally the restriction on link depth, which was intended to keep ring zero from getting into trouble, vanishes.

We can use this same mechanism of reflecting information out to an outer ring by setting a status code to indicate the fact that a segment's copy switch was set. This allows the concept of a copy switch to move out of ring zero. Whether it is still handled within the supervisor but in a higher ring or within the user's ring depends on whether it is to be considered a basic, unchangable system function or not. Personally I would move it to the user ring!

To complete our new ring zero address space manager interface we must introduce a terminate primitive. This primitive accepts three arguments. The first argument specifies the segment number to be terminated. The second argument specifies whether or not the released segment number is to be reserved. The final argument is a status code. It should be noticed that this primitive may be called with either a segment or directory segment number. In the case of terminating a directory one constraint is enforced. Since the system requires that a known segment's parent also be known, terminate will not terminate a directory with known inferiors.

Since this scheme removes the important function of name space management from ring zero we must provide a name space manager in the outer ring. Again it is a matter of opinion whether name space management should be handled in the supervisor or in the user ring. If it resides in the supervisor it cannot be clobbered by the user -- neither can it be changed. It is my opinion that it should reside in the user ring. Perhaps the system could also provide a secure address space manager which could be used by those users not interested in providing their own. I will assume that name space management will be moved to the user ring. Regardless of where it is placed all ring zero primitives which currently accept pathnames will have to become write arounds in some outer ring. These write arounds must first call an outer ring procedure which, through appropriate calls to the outer ring name space manager and the new ring zero address space primitives, translate pathnames into segment numbers. This corresponds to the function now performed in ring zero by find_. These segment numbers may then be passed to the new ring zero primitives which will not accept pathnames.

So far everything seems rosey. This scheme seems to remove many functions from ring zero and to simplify the ring zero interface in the bargain. Where is the hitch? Do we get all this for free? The answer is, of course, no. I have glossed over one important point. In order to decouple directory and segment initiation we must be able to sucessfully cloak the physical initiation of directories from a process' detection until it has established its right to know of the existence of the directory. As was pointed out earlier, this need for deception is intrinsic to the hierarchy structure and functionality of the current system. While this proposal makes the system's need to deceive the user more obvious, it is not responsible for the required deceit.

I will call a directory detectable if a process has established its right to know that the directory exists. Detectability may be established either by having non-null access to the directory or by having non-null access to its parent or by establishing the detectability of an inferior of the directory. The reason that non-null access on the parent of a branch establishes detectability is that either status , modify or

append permission is sufficient to allow the process to detect if the branch in question actually exists. It should be noted that the detectability of a directory is a function of the process' history and the ring of execution. A directory is detectable by a process in rings zero through the highest ring in which it has detectably initiated some member of the tree rooted at that directory. This highest detectable ring number of a directory is kept in its KSTE.

We must prevent a process from detecting any difference between an initiated directory which does not exist and an initiated existing but undetectable directory. If a process could detect a difference in these two cases then it could establish the existence of <u>any</u> postulated path in the hierarchy. This would constitute a clear violation of security. To accomplish this means abandoning the current one-to-one and onto mapping which exists between occupied segment numbers and known segments and directories. We must allow multiple segment numbers for the same directory. The reason for this is simple. Since the ACL of a segment completely controls the right to initiate that segment there is no need to allow a process to initiate a segment to which it has no access. This allows us to hide the physical existence of a segment from a process which has no right to know if the segment exists by returning the ambiguous status code noinfo in response to an initiate request. This simple mechanism fails for directories since we must always allow a process to initiate an existing directory in case it has access to some inferior of that directory. This forces us to return more than one segment number for a directory in some cases in order to prevent the process from detecting the existence of physically initiated but logically undetectable directories. If initiate_ returned the same segment number for two different entries then the process could be assured that the corresponding directory exists! This requires that we return a new segment number if a process reinitiates a directory which is still undetectable with a new name. In fact we will even return a new segment number if it tries to initiate an undetectable directory with the same name twice. If we returned the same segment number then inorder for directories which do not physically exist to appear the same to the user ring, ring zero would have to remember the name of every phoney directory. This is a needless complication of ring zero.

This scheme will merrily allow a process to initiate vast trees of directories which do not exist! These directories will be indistinguishable from real undetectable directories. The potential multiplicity of segment numbers for directories implies that if we compare two directory pointers and find them to be not equal we cannot conclude that the objects pointed to are not one and the same. Since processes running outside the supervisor cannot currently use segment numbers for directories, no user code can be effected by this new restriction. To allow processes to quickly determine if two segment numbers are bound to the same object the system should support a function for

mapping a segment number into the unique identifier of the object it is bound to. Naturally, this function must return an error if the object is not detectable to the process. The system must also insure that if the user attempts to reference through any directory pointer in an outer ring he will get the appropriate access violation whether our not the segment number he used corresponded to a real or phoney directory.

The action to be taken by ring zero in response to a request to initiate a directory depends on four boolean state variables of the target with respect to the accessing process. These variables can be encoded as a bit string with the interpretation of each bit given below.

state codes

state	meaning
1000	target's parent is phoney
0100	target detectable
0010	target exists
0001	target already has KSTE

The possible actions which ring zero can take in response to a request to initiate a directory are encoded below. I have omitted the case where the target is a link as this case has already been discussed.

action codes

aas	assign a segment number to the directory
ene	return a status code indicating that the directory does not exist
eno	return a status code indicating that the directory either does not exist or that the process has not established its right to know that it exists
rps	return segment number and a status code indicating that the directory was already known
sd	update highest detectable ring field of this KSTE and its superior KSTEs to the maximum of their current value and the ring of execution
sdz	set highest detectable ring field to zero

This encoding allows us to compactly characterize the functioning of initiate_ in the following table. Entries in the state column encode a possible state. Entries in the action column encode the actions to be taken given the state represented in the state column.

action of initiate_

state	action
00--	aas,sdz,end
010-	ene
0110	aas,sd
0111	rps
1---	aas,sdz,end

Two possible objections I can see to this scheme are that it can potentially waste segment numbers and it requires inspecting the parent's ACL. A close examination of the preceeding chart indicates that there are only two ways to assign

a segment number which is not directly connected to a directory. The first way is to reinitiate an undetectable directory. The second is to initiate a phoney directory. Neither of these operations should occur in normal operation. They could, however, arise in an attempt to use a misspelled pathname. To eradicate this problem the outer ring variant of find_ could terminate those directories which might be phoney if the terminal segment could not be initiated. This would prevent a habitual misspeller from cluttering up his address space. It seems that with this addition a process must go out of its way inorder to clutter up its address space. If that is what it wants fine! Even if a process wastes all its segment numbers it can recover by terminating no longer needed segment numbers. The apparent inefficency of inspecting the ACL of the parent of a branch during initiation of that branch is not serious since it is normally not required. Only when a process has null access to a branch and has not previously established detectability for that branch is it necessary to inspect the ACL of the parent.

In the old KST scheme, the names stored with each KSTE provided a means of telling what rings still had the associated segment or directory initiated. Since these names will no longer be kept in the KST some new mechanism must be invented to supply this information. This is easily accomplished by adding an eight bit field, called rings, to each KSTE. If the i th bit(0 origined) in this field is on then the corresponding ring has the segment or directory initiated. This allows ring zero to detect when a segment or directory may be physically terminated, thereby preventing one ring from terminating a segment or directory that is being used by another ring.

It should be carefully noted that the termination primitive terminates a segment number. Only if the last segment number for a directory is being terminated and its inferior count is zero will it be physically terminated! We can use the same method to describe the action of the terminate primitive as was used to describe the action of the initiate primitive.

state codes

state	meaning
100	KSTE has inferiors known
010	KSTE known in other rings
001	reserve requested

action codes

rr	reset this ring's known bit
tf	thread KSTE onto free chain
tr	thread KSTE onto reserved chain

action of terminate primitive

state	action
000	rr,tf
001	rr,tr
-1-	rr
1--	rr

In summary, this proposal calls for the complete removal of name space management from ring zero. As a result the concepts of pathname and file system links also depart ring zero. In the process of removing name space management from ring zero, I have reorganized and improved the ring zero interface and address space manager. The KST has been simplified and contains only two components: a KSTE array, and a UID hash table. The contents of each KSTE and their major uses are summarized below.

KSTE field	Use
forward pointer, backward pointer	Used to thread KSTE onto free or hash class list as required.
unique identifier	Unchanged (a phoney directory will have a uid = 0).
inferior count	Unchanged.
entry pointer	A packed pointer to the directory entry of this branch.
directory switch	Unchanged.
transparent modification switch, transparent usage switch	Unchanged.

rings	An eight bit field containing one bit per ring. Whenever ring i has this segment number initiated then bit i of this field is on.
highest detectable ring	A number which specifies the highest ring in which this process has established its right to know of the existence of this directory.

The proposed ring zero segment number manager interface is as follows.

initiate_ (dirsegno,ename,dirsw,rsw,link,segno,code)

dirsegno	segment number of the parent (input)
ename	entry name of target(input)
dirsw	directory switch(input)
rsw	reserved segment switch(input)
link	link(output)
segno	segment number of target(if rsw then input)
code	status code(output)

terminate_(segno,rsw,code)

segno	segment number to be terminated(input)
rsw	see above
code	see above

To help clarify the ideas presented in this proposal let us consider the following senario in which a process trys to initiate the segment >a>b>c>d>e>f in ring four. We will assume that directory e and segment f do not exist and that the process has no permission on a, b or d, and append permission on c in rings zero through four. To simplify matters we will ignore the existence of the outer ring name space manager and we will assume that we are operating in a virgin environment. What follows is how the outer ring fino_ would proceed in this case.

step 0 call initiate_(0,"",1,0,link,segno_of_root,code)

The root directory will be initiated, its detectble field in the KSTE will be set to four, and a status code of zero will be returned.

step 1 call
initiate_(segno_of_root,"a",1,0,link,segno_of_a,code)

The directory will be initiated, its detectable field in the KSTE will be set to four, and a status code of zero will be returned.

step 2 call initiate_(segno_of_a,"b",1,0,link,segno_of_b,code)

The directory will be initiated , its detectable field in the KSTE will be set to zero, and the status code noinfo will be returned.

step 3 call initiate_(segno_of_b,"c",1,0,link,segno_of_c,code)

The directory will be initiated, its detectable field in the KSTE will be set to four, and a zero status code will be returned. In addition this initiation establishes the process' right to know of the existence of superior directories at least in rings zero through four. This is reflected, in this case, by setting the detectable field in the KSTE of >a>b to four.

step 4 call initiate_(segno_of_c,"d",1,0,link,segno_of_d,code)

The directory d will be initiated, its detectable field in the KSTE will be set to four, and a zero status code will be returned.

step 5 call initiate_(segno_of_d,"e",1,0,link,segno_of_e,code)

The non existant directory e will be assigned a KSTE which will be marked as phoney and the status code noinfo will be returned.

step 6 call initiate_(segno_of_e,"f",0,0,link,segno_of_f,code)

No KSTE will be assigned and the status code noinfo will be returned.

step 7 call terminate_(segno_of_e,0,code)

The segment number assigned to e will be released on the grounds that e may really not exist.

The address space manager proposed in this MTB has been written and is many times simpler and smaller than the current ring zero address space manager. In some modules the reduction in size is on the order of a factor of ten! In addition, a version of hardcore which preserves the current ring zero interface is being debugged which is built on this new address space manager.

APPENDIX A

The main data base for the current ring zero address and name space manager is the Known Segment Table. The KST is a per-process, ring zero segment. Logically it contains four items. First, it contains an array of KST Entries. KSTEs are indexed by segment number and contain all per-process information necessary for the proper care and feeding of the segment or directory associated with the indexing segment number. Second, it contains a hash coded mapping from the space of Unique IDentifiers onto the space of segment numbers, or equivalently the space of KSTEs. This mapping provides the means of locating the KSTE of an already initiated segment should it subsequently be initiated by a different name. Third, it contains a hash coded mapping from the space of names onto the space of segment numbers. This association is mainly of use to the dynamic linking mechanism. Forth, it provides a repository for per-ring search rules. This later KST function will be considered no further as the user-ring dynamic linker removes this information from the KST. The current contents of a KSTE and their major usages are given in the following table.

KSTE field	Use
forward pointer, backward pointer	Used to chain the KSTE onto a list of free or reserved KSTEs as required.
unique identifier	Used to validate UID hash searches and to properly identify the corresponding branch after an on-line salvage.
name pointer	Used to chain together a list of the reference names associated with this segment or directory and the rings in which they are known.
inferior count	Used to prevent a directory from being terminated while it has known sons. If this were not done segment faults would fail!
parent segment number	Used at segment fault time to locate this branch's parent. It also is used to translate segment numbers into pathnames.
offset of branch	Used to locate the branch within the parent directory.
directory switch	Used to special case access setting for directories at segment fault time.
transparent modification, transparent usage switch	Used to control whether this process' usage and/or modification of this segment or directory should be transparent to the system.