PROJECT MAC

April 9, 1974

Computer Systems Research Division

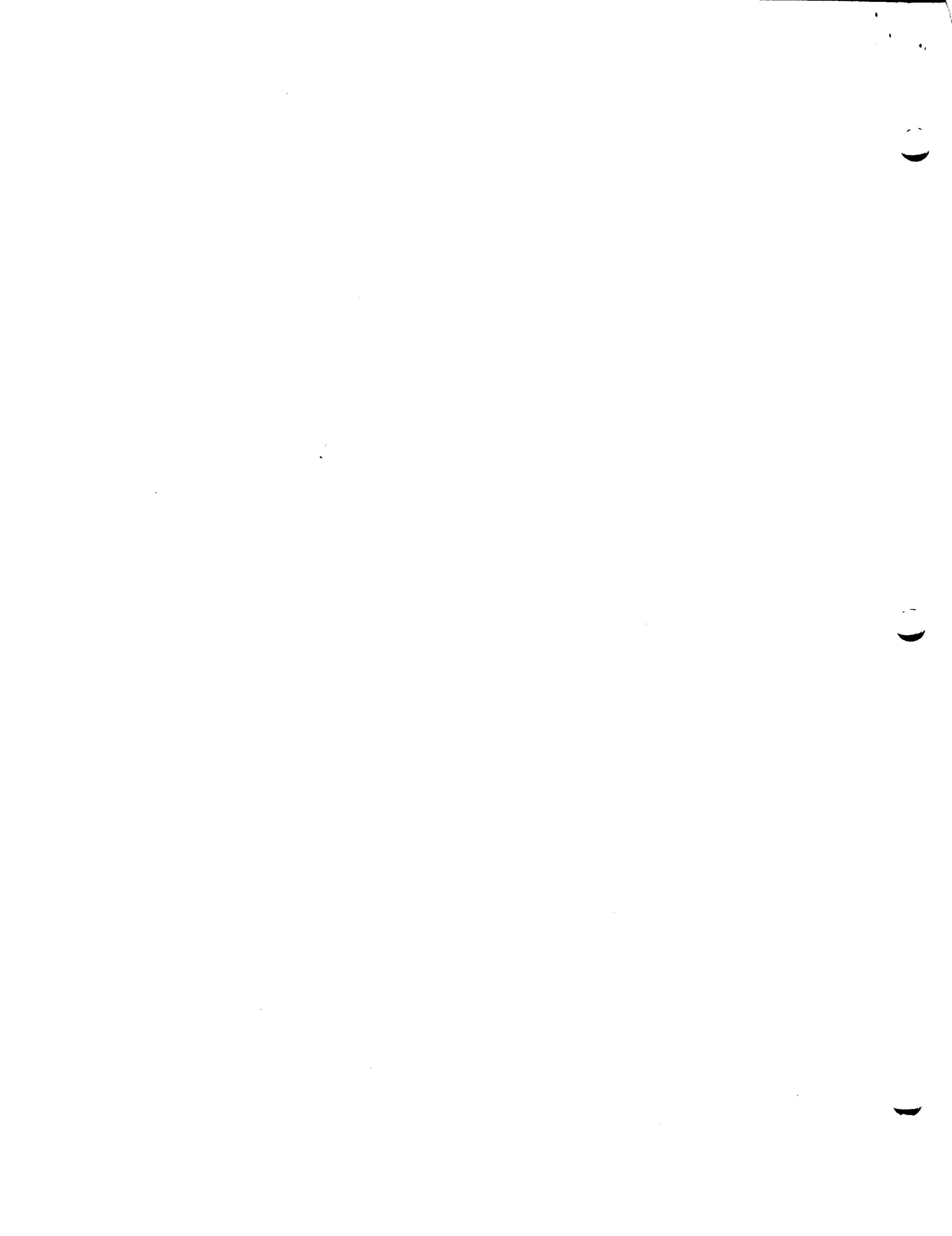
Request for Comments No. 49

A PROPOSAL FOR REMOVING NAME SPACE MANAGEMENT FROM RING ZERO

by Richard G. Bratt

This RFC, which is a first cut at a thesis proposal, describes a design for removing name space management from ring zero. The design described has the interesting property that directories may be directly initiated by a process. This leads to a very clean interface to the ring zero address space manager. In addition, it allows the processing of links and the resolution of pathnames to segment numbers to be handled in the outer rings.

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



Work has already begun to move non-security related system components out of ring zero. The hope is that as more and more extraneous functions are removed from ring zero it will approach a reasonable, certifiable primitive security kernel. This incremental approach means that we may not end up with as coherent a kernel as might be obtained by starting from scratch. However, it does have the strength of being constantly usable throughout its development. Phil Janson has already completed the design of a dynamic linker which runs in the outer rings requiring minimal ring zero support. This proposal suggests a design which removes the function of name space management(and some related functions) from ring zero.

Name space management consists of controlling the binding of names to objects in the environment. In the context of Multics , this proposal will consider name space management to be the controlling of the binding of names to segment numbers. This indirect binding is necessary because of a "defect" in the Multics hardware base. The hardware cannot represent an arbitrary object in a reasonably sized Multics hierarchy in its segment number registers and instruction fields. This requires that the objects which a process wishes to address first be mapped onto the space of segment numbers. This is accomplished by a process we will call address space management. Since processes tend to save segment numbers it would be untenable for the security kernel to attempt to simulate a machine in which all objects of interest could simultaneously be accessible. We therefore have two distinct steps in binding a name to an object. First, we must invoke address space management primitives to bind a segment number to an object. Second, we must invoke name space management primitives to bind a name to that segment number. Likewise, to completely reverse a binding of name to object two steps are required. Both of these functions are currently performed in ring zero.

The objects in the Multics environment with which name space management is concerned are segments and directories. These objects are organized into a single system-wide hierarchy (several special objects such as the system controller addressing segment are not members of this hierarchy). In point of fact, segments are the only really primitive objects in the Multics environment. Directories are merely segments which are only directly accessible in ring zero. This allows the ring zero directory control primitives to exercise complete interpretive control over access to directories. It can be argued that directory control need not reside in ring zero. It could just as well be located in ring one or perhaps in the user ring! This would effect a tremendous reduction in the size of ring zero. This proposal will not deal with such a removal of directory control from ring zero. The reason that I choose to leave directories in ring zero, for now, is that the access control list and physical storage map of a segment are stored in its containing directory. To move directory control out of ring zero

would involve moving these items out of directories. This would involve a major overhaul of the system, backup, and the salvager. Rather than make a change of this magnitude while moving name space management from ring zero, it seems reasonable to proceed one step at a time. After name space management has been sucessfully decoupled from address space management and removed from ring zero we can investigate the more complex issue of removing directory control from ring zero. Our primitive security kernel abstract machine, therefore, will support,for this proposal, the concept of a hierarchy of directories and segments.

Before describing my proposal, I will briefly list some of the current address and name space management mechanisms. These mechanisms all reside in ring zero.

Address Space Management

* functions explicitly accessible to outer rings
    * assign an available segment number to a segment
    * assign a reserved segment number to a segment
    * render a segment or directory inaccessible and make its segment number available for reassignment
    * render a segment or directory inaccessible and reserve its segment number
* functions implicitly accessible to outer rings
    * assign an available segment number to a directory

Name Space Management

* functions explicitly accessible to outer rings
    * associate a name with a segment's segment number
    * list the names associated with a segment number
    * disassociate a name from a segment number
    * get a pathname for a segment number
* functions implicitly accessible to outer rings
    * associate a currently valid pathname with a directory's segment number

The main data base for these address and name space management functions 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 new

dynamic linker proposal 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.

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. Names

of directories are constrained by ring zero to be absolute pathnames of the directory.

I submit that there should be no distinction between segment reference names and directory reference names. A process should be free to associate any name it chooses with a directory. Let us be uniform. We special case directories too often. Consider how easily the working directory and search directory concepts fit into such a scheme. All we need do is control the binding of reserved names such as: "working_dir" and "search_dir_n"!

It has been argued that the removal of reference names or directory path names from ring zero has the serious consequence that these names can no longer reflect name changes in the hierarchy (no such facility currently exists!). It seems obvious, to me, 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 must distinguish reference names from directory entry names. A reference name is only a name we temporarily assign 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 something 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 streches. If a process were to stay permanently logged in it would required 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.

The removal of name space management from ring zero obsoletes the (directory pathname, entry name) interface to address space control. This proposal suggests a radical change in the ring zero address space manager. I propose that processes be allowed to initiate directories directly. Currently there exists no means of determining whether a process should be permitted to initiate an arbitrary directory without searching the entire subtree rooted at that directory. This problem 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 the segments it has access to regardless of its access to these superiors! However, we also wish to prevent a process from detecting whether or not a given directory exists unless it has access to some inferior of that

directory. This is desirable since names of directories can potentially carry a very high information content. To avoid having to search the subtree of the hierarchy rooted at a particular directory to determine if the process should be permitted to initiate that directory, 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 to always refer to directories by pathname which is quite inefficient. If we allow ourselves to have pointers to segments why not for directories also?

This scheme of coupling directory and segment initiation has another drawback. Since a process cannot read the access control list of a segment until its parent is known, the system 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. The problem is, of course, that if a process can cause the system to malfunction if and only if it performs an operation which it must pretend not to perform to protect information, then it can cause the system 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 take a very different 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 the segment number of the root directory will have to be agreed upon by all processes. Initiation of segments will be controlled by

initiate_seg. Initiation of directories will be controlled by initiate_dir. The rationale behind providing seperate primitives for 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.

These new ring zero primitives accept identical arguments. The first argument is the segment number of a directory. The second argument is the name of the entry in the directory which is to be initiated. The third argument is the segment number of the target. If the forth argument, which is a reserved switch, is "1"b then the segment number is input otherwise an available segment number is assigned if necessary and returned in the segment number argument. The final argument is a file system status code.

An important consequence of not handling pathnames in ring zero is that links can no longer be interpreted in ring zero. When initiate_seg or initiate_dir encounter a link they must return a status code. This code informs the outer ring procedure that a link was encountered. It must then call a new ring zero primitive to read the contents of the link. This procedure, called get_link, has arguments similar to initiate_seg except that the third and fourth arguments are replaced by a character string variable in which the link is returned. Having read the link the outer ring procedure may then try the new path which it contains. 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 patnames.

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 either be able to tell, at a glance, if a process has the right to initiate a given directory, or 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. This right may be established either by having non-null access to the directory or by having non-null access to its parent or by initiating an inferior of the directory to which the process has non-null access. The reason that non-null access on the parent of a branch establishes a process' right to know of the existence of that branch is that either status permission or append permission is sufficient to allow the process to detect if the branch in question actually exists. Since Multics does not allow an ACL entry to contain modify permission without status permission any non-null access to a directory allows a process to determine the existence of a given son of that directory. I will call a directory detectable if a process has established its right to know that the directory exists. It should be noted that the value of this attibute 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 the process has non-null access to the parent of that directory or to some initiated member of the tree rooted at that directory. This highest detectable ring number of a directory is kept in its KSTE. After extensive investigation it seems that the most appropiate strategy involves cloaking the physical initiation of directories.

What we wish to do is prevent a process from detecting any difference in the initiation of an inferior segment or directory which does not exist and the initiation of an existing but undetectable inferior segment or 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. This mapping will remain one-to-one and onto for segments, however it must become an onto mapping for directories. 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 does_not_exist_or_no_access in response to an initiate request. This simple mechanism fails for directories since we must always allow a process to initiate an existing directory incase 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_dir 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. When multiple segment numbers are assigned to a real directory only one segment number, designated as the primary segment number, is actually used by ring zero to read and write the directory. The system will 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 was primary or secondary or corresponded to a phoney directory. Whenever the user passes directory control a segment number as a directory specifier it will first translate secondary segment numbers into the corresponding primary segment number. This is accomplished by adding a primary bit to every KSTE to determine whether it is primary or secondary. The forward pointer of a KSTE will point to the primary KSTE if its primary bit is off. The backward pointer of the KSTE, if non zero, will thread together, in a circular list, all KSTEs associated with a given directory.

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

<u>state codes</u>

<u>state</u>	<u>meaning</u>
10000	parent is phoney
01000	non-null perrission on parent in calling ring
00100	target exists
00010	target already has KSTE
00001	target detectable in calling ring

The possible actions which ring zero can take in response to a request to initiate a directory are encoded below.

action codes

aas	assign a secondary segment number and chain to primary KSTE
ans	assign a new primary segment number
ene	return a status code indicating that the directory does not exist
end	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
msd	update highest detectable ring field of superior KSTEs to the maximum of their current value and the highest detectable ring field calculated for the target KSTE
rps	return primary segment number and a status code indicating that the directory was already known
sd	set highest detectable ring field of the target KSTE to the maximum of the highest ring in which the process has non-null permission on the parent, and the highest ring in which the process has non-null access to the target
sdz	set highest detectable ring field to zero

The encoding of the relevent state of the system and the possible actions to be taken by initiate_dir allows us to compactly characterize the functioning of initiate_dir 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 dir

state	action
000--	ans,sdz,end
00100	ans,sdz,end
00101	ans,sd,msd
00110	aas,sdz,end
00111	rps
010--	ene
0110-	ans,sd,msd
0111-	rps
1----	ans,sdz,end

The only 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 initiate_dir will return a does_not_exist_or_no_access status code if it initiates a phoney or undetectable directory. These status codes will be ignored by the outer ring variant of find_ unless the terminal segment cannot be initiated. In this case the procedure could terminate those directories which might be phoney. 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 will the directory 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_dir primitive.

state codes

state	meaning
1000	KSTE is primary
0100	KSTE has secondaries
0010	KSTE known in other rings
0001	reserve requested

action codes

np	make first secondary KSTE the new primary
rn	return known in other rings error
rr	reset this ring's known bit
rt	rethread secondary chain
tf	thread KSTE onto free chain
tr	thread KSTE onto reserved chain

action of terminate primitive

state	action
0-00	tf,rt
0-01	tr,rt
0-1-	rn,rr
1000	tf
1001	tr
101-	rn,rr
1100	np,tf,rt
1101	np,tr,rt
111-	rn,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 address space manager. The important new ring zero interfaces are summarized below.

initiate_dir	initiate a directory
initiate_seg	initiate a segment
get_link	retrieve a file system link
terminate	terminate a segment number

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	Used to thread KSTE onto free or reserved list as required. When KSTE is assigned and it is marked as secondary this points to the primary KSTE for the associated directory.
backward pointer	Used to thread KSTE onto the free or reserved list as required. When the KSTE is assigned this field chains together all associated KSTEs. If it is zero this is the only KSTE for the given directory.
unique identifier	Unchanged.
inferior count	Unchanged.
parent segment number	Unchanged.
offset of branch	Unchanged.
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.
phoney	A one bit field which is set if this KSTE corresponds to a fictional directory.
primary	A one bit field which is set if this KSTE is a primary KSTE.
detectable	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_seg(dirsegno,ename,rsw,segno,code)
initiate_dir(dirsegno,ename,rsw,segno,code)
          dirsegno  segment number of the parent ( input)
          ename     entry name of target(input)
          rsw       reserved segment switch(input)
          segno     segment number of target(if rsw then input )
          code      status code(output)

get_link(dirsegno,ename,link,code)
          dirsegno  see above
          ename     see above
          link      file system link(output)
          code      see above

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 perrission on c in rings zero through five. 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 find_ would proceed in this case.

step 1 call initiate_dir(0,"a",segno_of_a,0,code)

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

step 2 call initiate_dir(segno_of_a,"b",segno_of_b,0,code)

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

step 3 call initiate_dir(segno_of_b,"c",segno_of_c,0,code)

The directory will be initiated, its detectable field in the KSTE will be set to five, 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 five. This is reflected, in this case, by setting the detectable field in the KSTEs of >a and >a>b to five.

step 4 call initiate_dir(segno_of_c,"d",segno_of_d,0,code)

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

step 5 call initiate_dir(segno_of_d,"e",segno_of_e,0,code)

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

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

No KSTE will be assigned and the status code does_not_exist_or_no_access 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.