M.I.T. LABORATORY FOR COMPUTER SCIENCE

March 31, 1977

Computer Systems Research Division

Request for Comments No. 140

# THE MULTICS KERNEL DESIGN PROJECT

by

Michael D. Schroeder*
David D. Clark
Jerome H. Saltzer

Massachusetts Institute of Technology
Laboratory for Computer Science

Draft of March 31, 1977

Please address correspondence to:

J. H. Saltzer
Massachusetts Institute of Technology
Room NE43-505
545 Technology Square
Cambridge, Mass. 02139

Telephone (617) 253-6016

---

This is a draft of a paper prepared for submission to the Sixth ACM Symposium on Operating Systems Principles, to be held at Purdue University, W. Lafayette, Indiana, November 16-18, 1977.

This research was performed in the Computer Systems Research Division of the M.I.T. Laboratory for Computer Science. It was sponsored in part by Honeywell Information Systems Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641, which was monitored by ISTAO under contract No. F19628-74-C-0193.

* Present affiliation of M. D. Schroeder: Xerox Palo Alto Research Center, Palo Alto, California.

This note is an informal working paper of the M.I.T. Laboratory for Computer Science, Computer Systems Research Division. It should not be reproduced without the author's permission and it should not be cited in other publications.

Abstract

We describe a plan to create an auditable version of Multics. The engineering experiments of that plan are now complete. Type extension as a design discipline has been demonstrated feasible, even for the internal workings of an operating system, where many subtle intermodule dependencies were discovered and controlled. Insight was gained on several tradeoffs between kernel complexity and user semantics. The performance and size effects of this work are encouraging. We conclude that verifiable operating system kernels may someday be feasible.

CR Categories: 2.12, 4.31, 4.32, 4.35, 6.21

Keywords and Phrases: Protection, Security, Security Kernel, Multics, Type Extension, Operating Systems, Supervisors, Verifiable Systems.

# THE MULTICS KERNEL DESIGN PROJECT

## Introduction

In 1974, a project was begun to apply the emerging ideas of security kernel technology, information flow control, and verification of correctness to a full function operating system, Multics. There were several aspects to this project; this paper discusses in depth the results of one aspect that was recently completed: some re-engineering experiments performed on the Multics supervisor to discover ways of simplifying it. To see how this part fits into the overall project, we first provide a project overview.

## The plan for a secure Multics

The version of Multics available in 1974 contained a wide variety of sophisticated security features, and it had been designed from the beginning (in 1965) with the integrity of those features as a goal. However, there were two problems from a security point of view. First, the set of programs that constituted the central supervisor and that could in principle compromise security contained some 54,000 lines of source code and had been touched by perhaps a hundred or more programmers during the development of the system. To do an integrity audit, one would have to examine and understand thoroughly every line of code in each of these programs. Although the programs in question were largely written in a higher-level language (PL/I) and were quite modular by function, auditing was still an overwhelming task. Second, the security mechanisms provided (access control lists with individual users, projects, rings of protection, passwords, etc.,) while useful, were somewhat

ad hoc, and did not fit into any simple underlying model. This lack of a simple model of security meant that even if an auditor were to undertake the previously mentioned overwhelming task of understanding every line of code, that auditor would lack a systematic specification of what to look for.

Yet, before one could entrust sensitive information to protection by an operating system, some kind of integrity audit seems essential. Therefore, a project was undertaken to make integrity auditing feasible, and to demonstrate that security is achievable in a large scale, full function operating system. As one might expect from the two problems mentioned, there were two key aspects to the project: 1) to simplify the supervisor so as to make it feasible for an integrity auditor to understand, and 2) to provide a set of security functions that can be described by a simple, understandable formal model. The overall plan was actually broken into several smaller components in order to allow an orderly experimentation and to take maximum advantage of already existing organizations. Figure 1 illustrates this plan.

The formal model used, because of its simplicity and apparent applicability to real world problems of the Air Force sponsor, is the MITRE model of sensitivity levels and compartments, which requires strict control of information flow among the levels and compartments [Bell and LaPadula, 1973]. The first step in this project (the box numbered 2 in figure 1) was to take the standard Multics system, and systematically add to it the particular set of security controls required by the MITRE model, which involved labelling all information with sensitivity level and compartment names, and adding security checks at all points where information could cross level or compartment boundaries. These changes resulted in a set of security features known as the Access Isolation Mechanism (AIM) and a version of Multics known as Multics

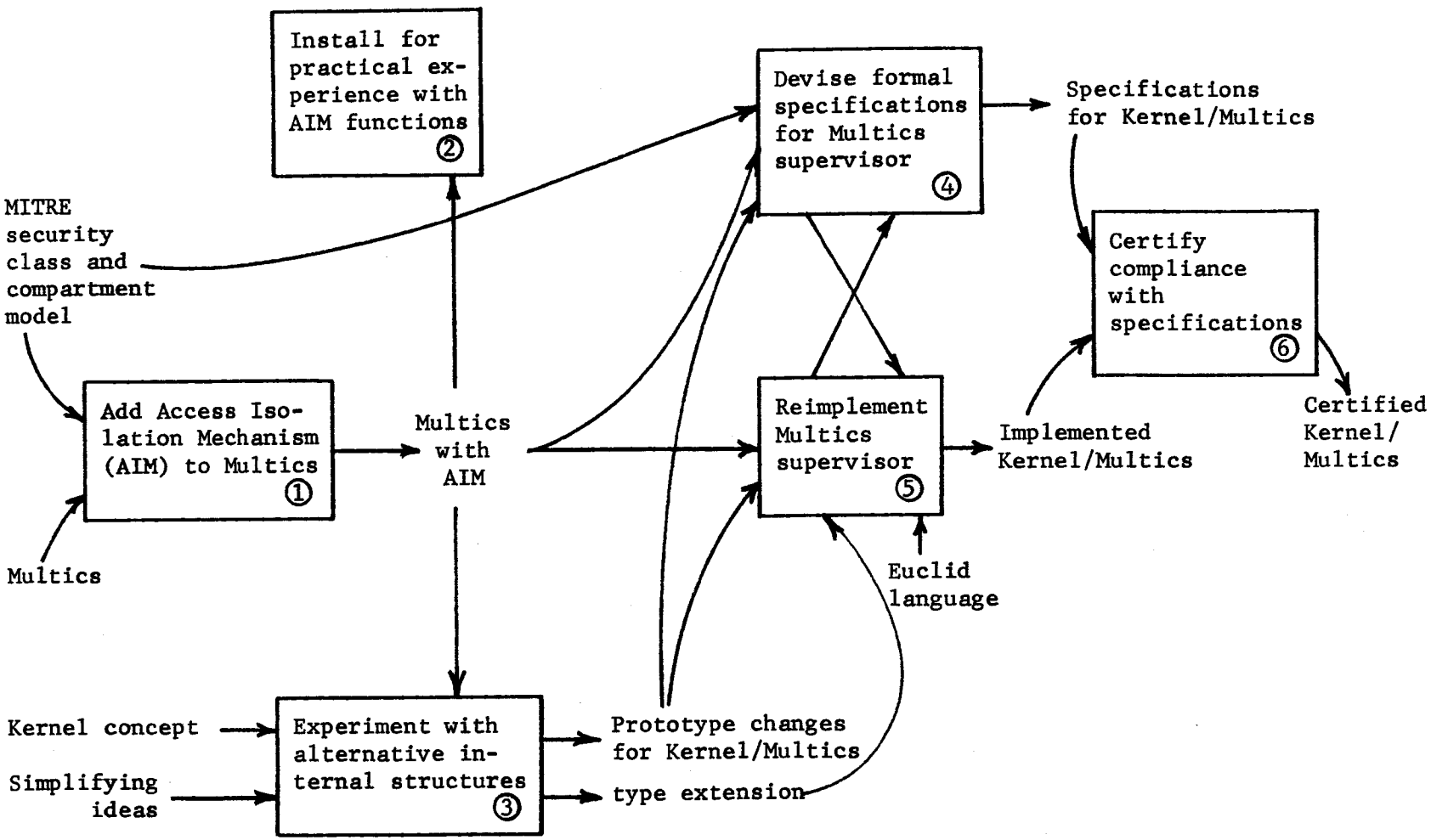



Figure 1 -- Plan for developing a certifiable security kernel for Multics

with AIM. Multics with AIM then became the base system for all future developments.

At this point, the work branched out in several directions. Multics with AIM was installed (box 2 of the figure) on a machine in the Air Force Data Services Center, and it was later made part of the standard product released to other Multics sites, so as to begin developing operational experience with the features of AIM. A series of prototype implementations were undertaken to discover ways of accomplishing the same functions with simpler and more systematic operating system structures with the discipline of the security kernel concept in mind (box 3 of the figure). And two groups of analysts began to develop successively more detailed examples of formal specifications of a kernel-based Multics with AIM, assuming the changes in structure proposed for experimental implementation turned out to be feasible (box 4 of the figure).

This description brings us to the stage of the Multics kernel design project today. The plan from here forward involves two major steps to be undertaken in parallel: first, the formal specifications of box 4 must be completed and they must be verified as matching the requirements of the MITRE security model. The second parallel step (box 5) is a reimplementation of the central supervisor of Multics, with two differences with the present implementation: those prototype simplifications that were successful will be incorporated, and the form of the implementation will be as "verifiable" as the state of the art will allow. This latter goal is to be aided by using type extension as a systematic programming discipline, and by using EUCLID [Lampson et al., 1976] as a programming language.

The result of these two efforts will be on the one hand, a new, easier-to-review implementation of Multics with AIM, to be known as Kernel/Multics, and on the other, a set of formal specifications traceable to the MITRE security model. The final step, box 6 of the figure, is unfortunately not as simple as its label suggests. A multipronged approach is proposed:

1) Program verification should be used wherever feasible. Although the state of the art of both automatic and manually assisted program verification technology for the foreseeable future is simply not yet capable of dealing with specifications and programs of the size and number involved in Kernel/Multics, formal verification may be applicable to some components.

2) Two or more small, expert teams of programmers can be assigned to be auditors of the code. With programs and specifications in hand, their job will be to try to understand the function of every program statement in Kernel/Multics, and to report anything that is not understandable or potentially in error.

3) The system can be placed in operational use. If the redesign has been successful, not only will security failures be prevented, but many other operating system reliability failures should not occur. Operational failures can be traced to see if they originate in the security kernel.

4) A tiger team can be assigned the task of breaking into the system.

Any one of these four approaches by itself cannot be expected to establish a credible verification of the integrity of Kernel/Multics, but the

hope is that the combination of all four in parallel can provide a much higher level of confidence in integrity than has ever before been achieved in a full-function general-purpose operating system. A second hope is that the tecnniques that are developed be applicable not just to Multics, but to other general-purpose operating system designs, and also to specialized systems that are dedicated to file storage and management.*

## Engineering studies for the Multics Kernel

As suggested, one of the key parts of this project was a series of prototype implementations of simplifying ideas for the kernel. An earlier paper [Schroeder, 1975] described the plans and justifications for these experiments, and reported results of some early restructuring that removed, wholesale, certain functions from the kernel. Without attempting to repeat that paper, the general strategy involved identifying all reasonable-sounding proposals for simplifying the Multics kernel, and then selecting for trial implementation those that could not be accepted as obviously straightforward or rejected as obviously inappropriate. Three kinds of redesign proposals emerged: 1) removing from the kernel those formerly protected supervisor functions that did not really require that protection; 2) taking advantage, whenever possible, of the natural protection afforded by independent processes communicating at arms length to implement protected functions, and 3) using more systematic program structuring techniques for implementing the remaining

* Several organizations have participated in this project. The overall plan was organized by the Air Force Electronics Systems Division. The AIM was implemented by Honeywell Information Systems Inc., with technical supervision from the MITRE Corporation. The M.I.T. Laboratory for Computer Science performed experiments with alternative structures. MITRE Corporation, and later SRI, devised successively more precise formal specifications for the Multics kernel. In October, 1976, with boxes 1, 2, and 3 of figure 1 completed, the Air Force suspended work on the project.

kernel function, so that the result might be easier to verify.

Probably the most interesting result of this work is the invention of a file system and processor multiplexing organization that is based on the discipline of type-extension, and that eliminates many complicating cycles of dependency in the kernel. This work required developing more carefully than usual analysis of the dependencies among supervisor modules, since the machinery of the type-extension implementation is itself part of the kernel.

The following sections of this paper describe briefly this type-extension system organization, several other structural results, and the estimated and observed effects of all these ideas on the size of the kernel and the performance of the overall operating system.

## Type extension as a rationale for coping with complexity

The initial projects of removing mechanisms from the Multics supervisor helped us understand what mechanisms needed to be present in a security kernel, but they did not help us understand how these pieces should be organized. To simplify the security kernel, it was important to develop an organizational rationale for modularizing the required functions and fitting them into an understandable overall structure. The rationale adopted is an application of the notion of type extension, and involves making all modules be object managers, categorizing all the ways one module can depend on another, and organizing the modules in a loop-free dependency structure. This rationale was developed by Janson and is reported in detail in his Ph.D. thesis [Janson, 1976]. Here we describe briefly this organizational technique and in the next section discuss its application to the Multics kernel.

Making each module be an object manager is a way of providing an understandable semantics for modules. The interface to a module defines all operations on the object type managed by that module, and thus defines the object type. Object types are chosen to have intuitive significance. For example, disk records, core blocks, core segments, page frames, active segments, and known segments are some of the object types used in the Multics kernel design. An object manager and the modules it depends on are solely responsible for maintaining the integrity of the managed objects. Client modules can manipulate the objects only through the interface provided by the object manager. Knowledge of the way an object type is represented is confined to the manager module. A representation is a set of lower level component objects and the algorithms relating the operations of the object type to those of its components.

When trying to develop an understanding of the way a collection of object manager modules works, the important consideration is the way the modules depend upon one another. One module depends upon another if establishing the correct operation of the first requires assuming the correct operation of the second. Requiring a loop-free dependency structure, i.e., requiring that the structure generated by the "depends on" relation between modules be a lattice, allows system correctness to be established iteratively, one module at a time.

Inside an operating system careful analysis is required to identify all intermodule dependencies. The opportunity exists for an operating system module to produce dependency loops by participating in the implementation of its own execution environment. Such opportunities are less of a problem for application programs, which typically depend on the operating system to provide their execution environments. To develop the complete dependency

structure of a collection of object manager modules in an operating system, five kinds of dependencies need to be considered for each module. For a module M the possible kinds of dependencies on other modules are:

a. Component Dependencies

Module M depends on the modules that manage the objects that are the components of the objects defined by M. For example, the manager of file system directory objects in the Multics kernel has a component dependency on the manager of segment objects, for each directory representation is stored in a segment.

b. Map Dependencies

Module M must maintain a mapping between the names of the objects it manages and the names of the components of each. Thus, M depends on the managers that provide the objects in which the map is stored.

c. Program Dependencies

The algorithms of M and their temporary storage are contained in objects, whose managers M thus depends on.

d. Address Space Dependencies

The address space in which M executes is an object, on whose manager M thus depends.

e. Interpreter Dependencies

In order to execute, M requires an interpreter, i.e., a virtual processor. Thus, M depends on the module that implements its interpeter.

This partition of dependencies into five categories is complete and fairly intuitive for systems designed according to the rationale of type extension. When applied to an existing design that was modularized and structured by different principles (or no principles at all!) one can encounter explicit dependencies due to procedure calls or due to interprocess messages from which replies are expected and implicit dependencies, due to direct sharing of writable data among modules, that do not fit naturally into this classification. The proper classification of such dependencies is of no concern, however, since the goal is their elimination and evolution to a design in which all dependencies fit naturally into this scheme.

Using the rationale just described, and with the five kinds of dependencies in mind, it was possible to design a loop-free structure of object managers that implement the complete functionality required in the Multics kernel. Our experience in doing so is described in the next section.

## Get the loops out

The file system, memory management, and processor management portions of the supervisor of Multics (which together constitute the bulk of the supervisor) appear to be organized in the six large modules illustrated in Figure 2. The obvious exception to a linear structure is the circular dependency of the processor multiplexing facilities and the virtual memory mechanism. (Page control depends upon process control to give the processor to another process when the current process encounters a missing page exception. Process control in turn depends upon segment control to provide segments in which to store the states of inactive processes. Thus, for example, a missing page exception for process A causes page control to invoke

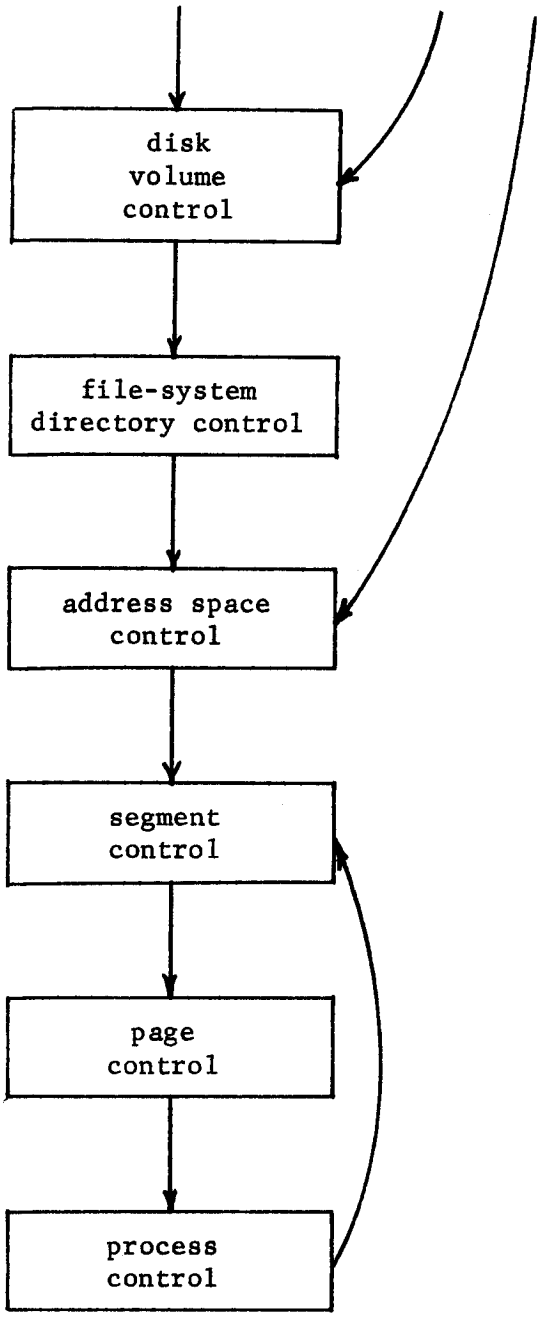



Figure 2 -- Superficial Dependency Structure
in Multics.

process control, which in turn invokes segment control to load the state of process B into primary memory using page control.) This dependency loop is common to many virtual memory time-sharing systems and is caused by the virtual memory mechanism being part of its own interpreter. In addition to this obvious dependency loop there are numerous examples of modules depending upon higher modules to contain their programs and maps, and represent their address spaces. For example, page control code is stored in segments and the address space in which page control executes is provided by address space control. Closer inspection reveals other loops in the dependency structure--all related to handling exceptional conditions or controlling resource usage. Simplified descriptions of several problems typical of these more subtle loops follow:

a. Missing Pages

Because Multics has multiple real processors, two processes simultaneously may cause page control to attempt to alter the state of the same page. A global lock prevents such conflicts. Unfortunately, the hardware imposes a short time window between a missing page exception and the setting of the lock by page control and some other process may alter the address translation tables between the exception and capturing the lock. Page control interpretively retranslates the virtual address that caused the exception once the lock is captured, to see if the absolute address of the page descriptor that resulted from the hardware translation causing the exception is still correct. This interpretive retranslation requires page control to know the format of and depend upon the correctness of the address translation tables maintained by segment control and address space control.

b. Quota Enforcement

Arbitrary directories in the hierarchy of file system directories can be designated dynamically as quota directories. Associated with a quota directory is a limit on the total number of pages that may be occupied by segments that are in the subtree below the quota directory but not also below an inferior quota directory. Also associated with a quota directory is a count of the total number of pages currently occupied by segments in the controlled region. Whenever a segment is to be grown, it is necessary to find the limit and count of the nearest superior quota directory, check that the count does not use all the limit, and if quota remains increment the count. The need to grow a segment is noticed in page control as a missing page exception on a never-before-used page of a segment. Before adding the page to the segment, page control must locate and manipulate the limit and count associated with the nearest superior quota directory, as described above. Thus, page control must identify the page with a segment and the segment with its position in the directory hierarchy. Page control does so by direct reference to the segment control data base, the active segment table, that associates each active segment with the descriptors for its component pages. The limit and count of a quota directory are kept in the entry of the segment representing that directory whenever that segment is active. Segment control is careful never to deactivate a segment that is a directory if inferior segments in the file system hierarchy are active. Further, segment control links each active segment table entry to the (always present) entry of its immediately superior directory. Thus, page control can locate the appropriate limit and count by following the links until

the entry of a quota directory is found. This implementation of quotas and storage usage records makes page control depend on segment control, and constrains segment control's management of the active segment table to follow the shape of the directory hierarchy defined by directory control.

c. Full Disk Packs

A file system directory entry in Multics names the corresponding segment by the identifier of the containing disk pack and an index into that pack's table of contents. For robustness and demountability, all pages of a segment are kept on the same pack. Thus, growing a segment occasionally causes a full pack exception, which results in the entire segment being moved to an emptier pack and the directory entry being updated to indicate the new location. The need to grow a segment is noticed by page control when processing a missing page exception for a never-before-used page of a segment. If a full disk pack exception is detected, page control invokes segment control, which directs the relocation effort. To accomplish relocation, segment control reads a data base maintained by address space control to find the corresponding directory entry, which segment control then directly updates.

Once the dependencies generated by these and similar causes are taken into account, the simple, almost linear structure of the system illustrated in Figure 2 becomes the much less simple structure illustrated in Figure 3.

The restructuring of the file system, memory management, and process management portions of the Multics supervisor that eliminates all dependency loops and provides an understandable object-based semantics for each module

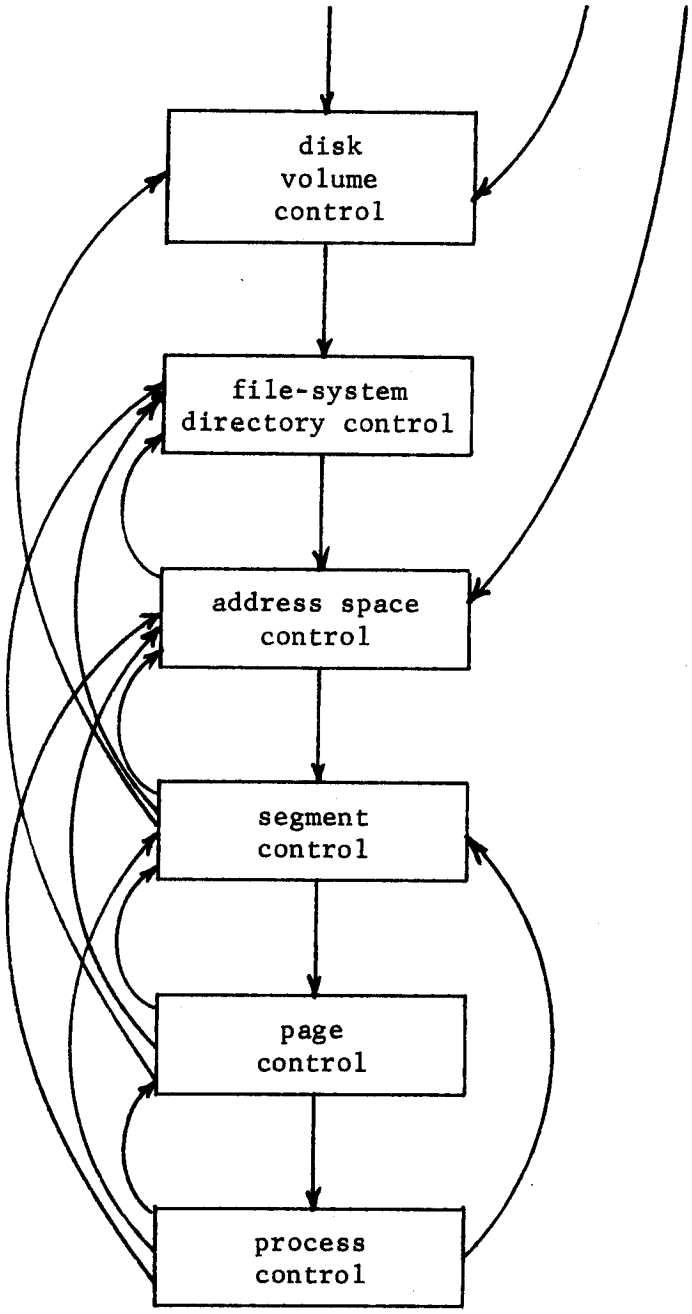
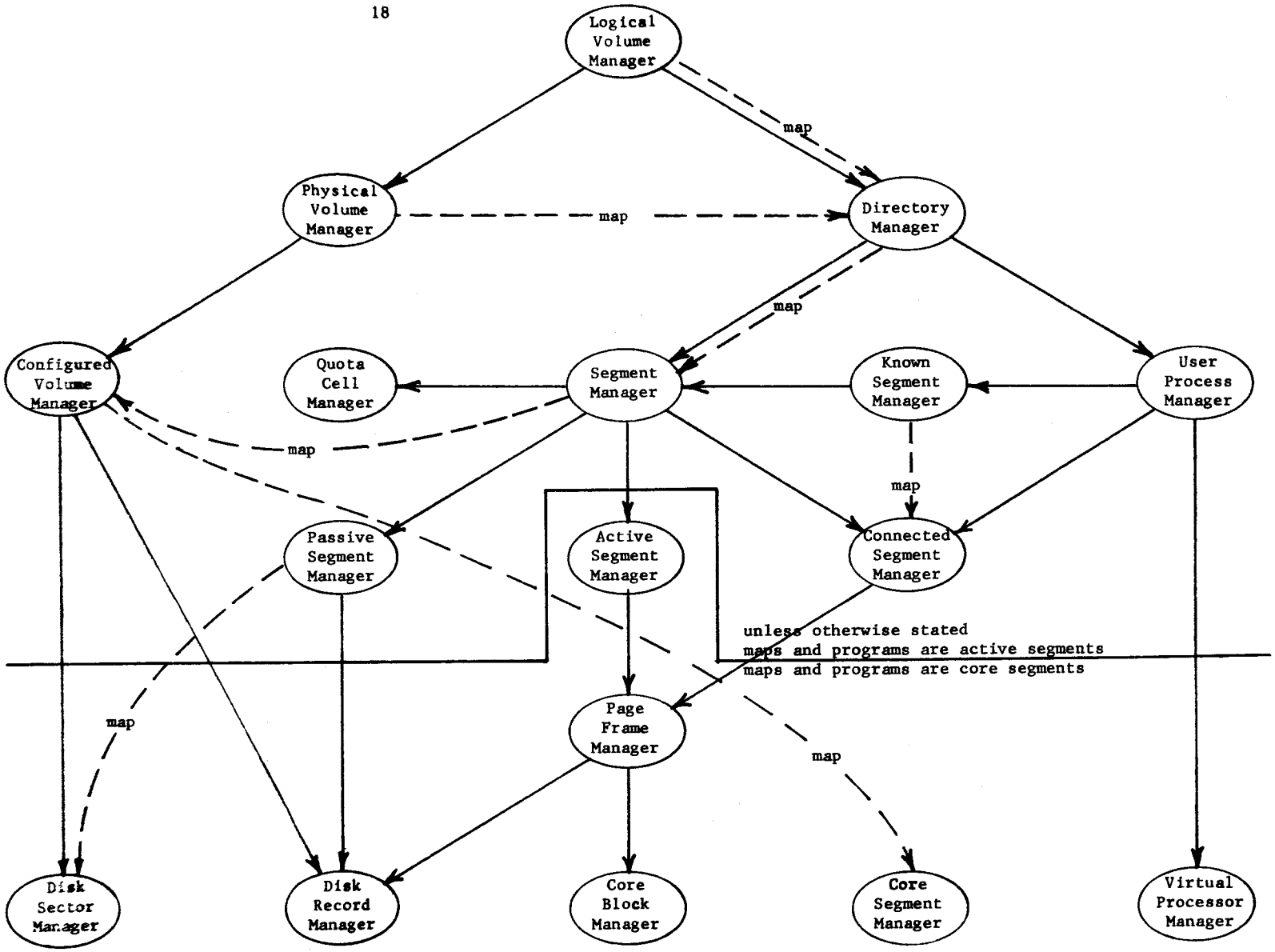



Figure 3 -- Actual Dependency Structure
in Multics

was worked out by Janson and Reed and is described in detail in their theses [Janson, 1976; Reed, 1976]. Here we indicate in general how the new design eliminates the structural problems outlined above, and make some comments on the causes and solutions of such problems in general. Figure 4, taken from Janson's thesis, shows the modules of their design and indicates the dependency relationships among the modules.

The loop between the processor multiplexing facilities and the virtual memory mechanism is broken by dividing process control into two parts, the user process manager and the virtual processor manager illustrated in Figure 4. The bottom part implements a fixed number of virtual processors whose states are always in primary memory. Thus, this part does not need to use the virtual memory. The top part implements an arbitrary number of user processes and depends upon the virtual memory to store their states. A subset of the virtual processors are multiplexed among the user processes as needed. The remaining virtual processors are permanently bound to the interpretation of various kernel modules, including the virtual memory modules and the user process scheduler. Use of a two-level process implementation in a Multics kernel is worked out in sufficient detail that we are confident this design provides a practical, well-structured method for providing an arbitrary number of processes in a system with virtual memory. The two-level design also provides a general way to eliminate all loops created by interpreter dependencies, for the bottom level provides an interpreter that depends on only the primary memory and the hardware processors.

Loops due to map, program and address space dependencies are relatively easy to break once their existence is recognized. The key to breaking these loops in the new design is the explicit recognition of core segments as




interpreter dependencies: every module, except the Core Segment Manager, depends on the Virtual Processor Manager
address space dependencies: every module, except the Core Segment Manager, depends on the Core Segment Manager

objects. The core segment manager of Figure 4 is implemented by system initialization code and by the processor hardware. The core segments are allocated when the system is initialized and thereafter the only available operations on them are the processor read and write operations. A core segment can be used by any system module to contain maps or programs and their temporary storage without fear of creating a dependency loop. Use must be tempered, however, by the facts that the number of core segments is fixed, the size of a core segment cannot change, and core segments are permanently resident in primary memory. To eliminate address space dependency loops a second address translation table base register is added to the processor. One base register locates the address translation table, stored in a virtual memory segment, that defines the address space in which user programs execute, while the other locates a translation table, stored in a core segment, that defines a per processor address space for system modules.* In use, all segment descriptors in the latter translation table will be for permanently active segments, i.e., segments whose page descriptors are always in primary memory, or core segments. All segment numbers below a certain value are translated relative to the system module address space. Thus, system modules using these segment numbers cannot be dependent on the machinery that supports the users' virtual address spaces.

Correction of the dependency loop surrounding missing page exceptions requires an addition to the processor architecture. Recall that to eliminate potential conflicts over the offending page descriptor, page control must reinterpret the virtual address that caused the exception after a global lock

* An implementation without extra hardware is also feasible, though a bit clumsy and not so modular, by sharing the first page of all address translation tables.

is set. A simple processor addition that corrects this problem is a mechanism that sets a lock bit in the offending page descriptor whenever a descriptor is encountered that indicates a missing page. Once the lock is set control is transfered to the page frame manager of Figure 4. A processor encountering a locked page descriptor will generate a locked page descriptor exception that results in the page frame manager calling the wait primitive of the virtual processor manager. Once the original missing page exception is serviced, the page frame manager unlocks the descriptor and notifies all processes that have been waiting for this event, causing them to start execution again at the point just previous to encountering the locked page descriptor exception. In addition to the descriptor lock mechanism, a wakeup waiting switch and a register to record the absolute address of the locked page descriptor can be added to each processor to aid in preventing a notification from being lost if it occurs between a locked page descriptor exception and invocation of the wait primitive.

The solution to the dependency loops associated with quotas and full disk packs illustrate two alternative mechanisms for reporting exceptional conditions without creating dependencies. A problem common to both situations is that software in some module discovers after some processing that a condition exists that needs to be handled at a higher level in the dependency structure. As described earlier, the condition results either in the module directly referencing the data bases at the higher level, or in the module calling the higher level module. The two mechanisms that can break these dependency loops are hardware that directly detects the exceptional condition and invokes the higher level module in the first place, or software that transfers control and arguments to a higher level module without leaving

behind any procedure activation records or other unfinished business in expectation of a subsequent return of control. Both solutions are illustrated below.

In the case of quota enforcement and recording disk usage, recall that the need to grow a segment, and thus to check the associated quota, is noticed in page control as a missing page exception on a never-before-used page. The new design has the hardware distinguish such events and generate quota exceptions instead. The exception is distinguished by an extra exception-causing bit in page descriptors that is set by software when the descriptor corresponds to an unallocated page in a segment. The quota exception invokes the known segment manager of Figure 4, reporting the segment number and page number from the address whose translation caused the problem. The known segment manager translates the segment number into a segment unique identifier and invokes the segment manager to find the appropriate quota directory, check the limit, and then call the page frame manager to add the page to the segment. In addition to the hardware quota exception, the new design makes quota cells be explicit objects with their own manager, as indicated in Figure 4. A quota cell is stored in the disk pack table of contents entry for the associated directory and is cached in primary memory in a table managed by the quota cell manager. The segment manager presents the quota cell information to the quota cell manager whenever a directory is activated and calls upon the quota cell manager to perform all operations on quota cells. A slight change is made in the semantics of quota. The change limits the dynamic designation of a directory as a quota directory and the inverse operation to occur only if a directory has no children. Because of this change, the relationship between each segment and its controlling quota

directory becomes static, and a dynamic upward search of the hierarchy to locate the appropriate quota directory is no longer required each time a segment is grown. Whenever the known segment manager asks the segment manager to activate a segment, it provides the identity of the appropriate superior quota directory and the segment manager simply associates the static name of this directory's quota cell with the segment's identifier and presents this name to the quota cell manager whenever quota must be checked. As a result, the deactivation of segments by the active segment manager no longer is constrained by the shape of the directory hierarchy.

The loop associated with full disk packs is broken by the use of the software mechanism for upward signalling. A full disk pack occasionally is encountered when processing a quota exception. If quota exceptions, which are detected by the hardware as described above, all were signalled directly to the directory manager, then a relatively simple mechanism for dealing with full disk packs would result. The directory manager would initiate a chain of calls down through the dependency structure that allowed the known segment, segment, and page frame managers to play their parts in checking quota, recording usage, and allocating a page. Further, if the page frame manager at the end of this call chain noticed a full disk pack when attempting to add the page to the segment, then this exception could be returned back up the call chain, allowing the segment manager to disconnect all address spaces from the segment and direct its movement to another pack, and allowing the resulting new pack identifier and table of contents index to be returned to the directory manager for inclusion in the corresponding directory entry. Unfortunately, it is too inefficient to pass all quota exceptions to the directory manager just to handle easily the full disk pack exceptions that

very occasionally accompany them.

Another solution that would generate a simple software structure is for the hardware to separate quota exceptions that will involve full disk packs from those that will not, signalling the former to the directory manager and the latter to the known segment manager. But it is unreasonable to expect the hardware to make the separation in this complex case.

Thus, we must make do with all quota exceptions being signalled to the known segment manager, which initiates c chain of calls down through the de pendency structure to handle them. A full disk pack exception is detected at the bottom by the page frame manager, which exception is returned back up the call chain as described earlier. Control finally returns to the known segment manager with both the quota and the unsuspected full disk pack exceptions taken care of, and with the pack identifier and table of contents index that locate the moved segment. The problem now is for the known segment manager to cause the directory manager to update the corresponding directory entry with the new disk location for the segment. This problem is solved by using the new signalling mechanism to transfer control and arguments to the directory manager without leaving behind any procedure activation records. Thus, modules below the directory manager in the dependency structure do not depend on it finishing the job of updating the directory entry. When the directory manager completes updating the appropriate directory entry, it restores the user process to the state it had just before the original quota exception (a description of this state is also passed to the directory manager with the software signal), and the process then rereferences the segment. At this point all processes rereferencing the segment will be reconnected via the standard machinery for handling missing segment exceptions.

This completes the discussion of the structural problem found in Multics and the methods used to deal with them. In several cases the mechanisms described are somewhat simplified from those actually proposed for use in the kernel of a secure Multics. The extra complexity of the mechanisms actually proposed is the result of a desire to use an unmodified hardware base for the real system, or of an attempt to achieve better performance with more complex (but still well structured) mechanisms that short circuit several layers of name mapping. The primary purpose of this discussion has been to communicate the flavor of the problems encountered in a real system design, and illustrate the types of solutions found to these real problems. Extensive analysis of the kernel design will be found in the theses by Janson and Reed. Some related ideas concerning the use of object property lists to break dependency loops will be found in the thesis by Hunt [Hunt, 1976].

We summarize our experience in applying the type extension rationale to structuring the Multics kernel with the following observations. Most systems appear to have a loop-free dependency structure if viewed from far enough away. The obvious component relationships and the common operations follow loop-free paths among the modules. On close inspection, however, map, program, address space, and interpreter dependencies will almost certainly generate loops in a system designed without loop avoidance as a primary objective. The map, program and address space loops usually are broken easily (at least during the design stage) by introducing new object types to store the maps, programs, and address space definitions. The interpreter dependency loops appear to be eliminated in most systems by using a two level implementation of processes. The most difficult and subtle structural problems are caused by exception handling--especially when the exceptions are

part of the mechanisms that control resource usage. The difficulty is partly intrinsic--such exceptions tend to occur at low levels in the system but be related to high level objects--and partly methodological--resource usage controls and the paths followed to deal with exceptions tend to be added to a design last. A general method for removing loops related to exception handling and resource control is harder to see, but in many cases removal involves improvement of hardware exception reporting mechanisms or addition of software mechanisms for signalling upward in the dependency structure without generating new dependencies.

From simple semantics do complex implementations grow

Much of the complexity of a system implementation can arise from only a few of the features being implemented. When one realizes that a particular feature causes complexity, it is time to review the importance of the feature and to see if a slight variation in its semantics might lead to a simpler implementation. In the course of reviewing the mechanisms of Multics to see how they affected a kernel implementation, several examples of this phenomenon were noted, and a fair amount of insight into the implications of certain user-visible features was thereby acquired. One example, the dynamic designation of directories as repositories for disk storage quota, which leads to great complexity in managing active segments, has already been discussed. In this case, a slight change of semantics seemed worthwhile.

A second example of complexity arising from apparently simple semantics has to do with whether or not the number and identity of processes implemented by the system should be fixed or dynamically variable. Brinch Hansen has argued fairly convincingly that considerable simplification of implementation

follows a decision to implement a fixed number of processes [Brinch Hansen, 1975]. On the other hand, when one tries to open the dependency loop between process implementation and virtual memory implementation as mentioned before, every process state would have to be resident in the fastest, most expensive memory medium. If the number of processes is fixed at the maximum that would ever be needed, valuable primary memory space would be unused at other times.

This combination of pressures led to the design for a two-level implementation of processor multiplexing. Since in this design the number of first-level virtual processors is fixed, all the simplifying advantages suggested by Brinch Hansen occur. This strategy of a two-level process implementation has been proposed elsewhere [Bredt and Saxena, 1975; Neumann et al., 1975] but these other proposals have omitted a key complicating factor: events discovered by low-level virtual processors must be signalled to user level processes, and communicating such signals requires access to the state of the user-level receiving process, which state by design is not guaranteed to be in the real memory accessible to the low-level virtual processor. As part of the Multics kernel design, Reed developed a method for this upward communication that makes the two-level process implementation feasible. The design involves placing a special, real memory message queue between the lower-level and higher-level processor multiplexers [Reed, 1976]. It also involves using a new synchronizing protocol, based on eventcounts, that controls information flow between processes and does not require that the discoverer of an event have knowledge of the identity of the processes awaiting that event [Reed and Kanodia, 1977].

For another example of complicating semantics, a combination of access control and naming semantics in Multics conspires to force some remarkable

maneuvering inside the supervisor. The directories of the Multics storage system are arranged in a naming hierarchy, and every file and directory has its own access control list, which specifies who may use the file or directory. Directories have access control lists on the basis that the names of files (and other directories) often contain information, so access to those names should be controlled, too. Finally, to make the semantics of access control as simple as possible, the rule is made that access to a file is determined entirely by the access control list for that file. This rule means that if one user wishes to grant another user access to a file, the first user places the other user's name on the access control list of the file, and the transaction is complete, without need to revise or check access control lists of directories higher in the naming hierarchy.

Now, suppose a user presents the storage system with the tree name of some file deep in the hierarchy, and the tree name traverses one or more directories to which the user does not have access. The simplifying rule requires that the file system follow the name through those inaccessible directories in order to get to the access control list of the file. If access to the file is indeed permitted, that user will, by virtue of not getting an error message, confirm the existence and names of the intervening directory structure. On the other hand, if access to the file is not permitted, the file system must be very careful in its response so as not to confirm the file name, or the names of the intervening directories.

The non-kernel version of Multics handled this set of constraints by burying the entire directory search operation inside the supervisor, and reporting one of two responses: "file found", or "no access". (This last response offers no clue as to whether or not the file and the directories

corresponding to the presented name exist.) In attempting to reduce the size of the machinery that must be in the Multics kernel, it was apparent that the general operation of following path names did not need to be a protected mechanism. If the supervisor kernel provides a primitive to search a single, designated directory for a presented name, and it returns the identifier of any matching entry, the program that knows about how to expand tree names need not be in the supervisor. Except, of course, that the particular protection semantics in use require that the kernel not return the identifier of a matching entry unless either the directory is accessible to the user or the file ultimately to be addressed is accessible. The first case is easy, but the second one produces a problem.

An elegant, if unsatisfying, gimmick was invented by Bratt [Bratt, 1975] to finesse the problem. The directory searching primitive, if asked to search an inaccessible directory, always returns a matching identifier for the presented name, whether or not the name exists. It will even return an identifier if asked to search a non-existent directory. This returned identifier, if then presented as a directory identifier to the directory searching primitive, is always accepted. In the case that the path of directories eventually leads to a file to which the user has access, each of the intervening directory identifiers is real, as is the ultimately returned file identifier. If, however, the user does not have access to the object at the other end, his attempt to use this ultimate identifier will result in a "no access" response from the file system, and he will be unable to decide whether or not the identifier (and all those of inaccessible traversed directories) is real or mythical.

From a broader perspective, this interaction between protection and naming sematics seems to leave three choices: a bizarre interface, as just described, or implementing the entire function in the kernel (the earlier design), or varying the user-visible semantics of protection or naming. But the particular semantics in use were already the result of several years of experiments with different kinds of semantics, and the particular rules described have turned out to minimize errors and simplify user comprehension [Saltzer, CACM, 1974]. Getting all these considerations adjusted just right is an open problem. It seems likely that a more explicit separation of user-level semantics for naming and from those of protection, such as found in UNIX [Ritchie, 1974] would help.

An interesting final case study of tradeoff between implementation complexity and user interface semantics arises in the Multics treatment of secondary (disk) memory storage charges. The user interface calls for a charge for just the storage required to implement a file. Since page-sized blocks of zeros happen to be implemented by flags in the file map rather than by allocating and storing whole pages full of zeros, a file of size of say, 100,000 words (100 pages) but non-zero in only the first and last words will accumulate a charge for only two storage pages. Users have taken advantage of this feature to simplify many file-manipulating programs. They create from the beginning a file of the maximum size that might ever be needed, but for much of its life the file contains little data, so it costs little to store.

This policy has three effects on the complexity of the kernel of the operating system. First, any time the user writes data into a file, the number of pages required to implement the file may change, and thus the appropriate quota directory may need to be updated. Care is required to

implement this update without creating a dependency loop. Second, the page removal algorithm finds that part of its specification includes searching the contents of pages about to be removed, to see if all words are now zeros. Thus this algorithm must be given (otherwise unnecessary) access to the data in every page of every file stored by the system. Finally, since files are read by mapping them into blocks of core memory, if a user tries to read from a page containing all zeros, a zero containing page must be allocated, at least temporarily, and the accounting measures must be updated. Thus a read implicitly causes information to be written, perhaps on the other side of a protection boundary, in violation of the confinement goal [Lampson, 1973].

Naming-related storage quotas, variable numbers of processes, naming-related access control, and accounting for physical representation costs are typical examples of conflicts between desired semantics and implementation complexity that were encountered in the Multics kernel simplification effort. It is interesting to conjecture whether or not these conflicts would also arise in a computer system dedicated to just file storage and management. In such a system, one might successfully fix the number of processes (although we doubt it), but the other conflicts certainly remain.

Impact of engineering studies on the size of the Multics kernel

There are a variety of measures that can be used to assess the size of the Multics kernel. One can count the number of lines of source code, but this count is confused by the fact that while most of the code is written in PL/I, some is in assembly language. This distinction could be eliminated by counting the number of machine instructions in the kernel, but this number seems somewhat irrelevant, since no auditing procedure is likely to involve an

examination of the machine instruction themselves. The most useful and consistent measure of the kernel size seems to be the number of source lines that would exist had the system been coded uniformly in PL/I, and this is the measure we will use.

The largest component of the kernel is those programs that are within the innermost protection boundary of the supervisor, known locally as ring zero programs. At the beginning of this project there were the equivalent of 36,000 lines of PL/I within ring zero. As some measure of the modularity of this code, there existed approximately 1,200 distinct entry points in the supervisor, of which 157 were callable by the user. In addition to the ring zero programs, there are a number of other programs that ought to be included as part of the Multics kernel: there were programs in other supervisor rings, and there were also programs that ran in trusted processes. One study was made of the largest of of these non-ring zero programs: the Answering Service, the programs that regulate attempts to log in to the system, including authenticating passwords, and manage system accounting. These programs were the equivalent of 10,000 lines of PL/I code. It is clear that the non-ring zero programs contribute significant bulk to the kernel of the system.

It is interesting to observe the effects which have occurred in the standard system between the time of that first census, in September of 1973, and the present time. During that time, the size of both ring zero and the next outer ring, both of which need to be considered part of the kernel, have almost doubled in size. There are a variety of effects that have contributed to this growth, primarily more sophisticated detection of coping with errors, and also some new functions.

As mentioned above, some of the kernel, approximately 10%, is coded in assembly language rather than PL/I. Because of this, the number of source lines in ring zero is actually not 36,000 but 44,000. Thus, there would be a substantial size benefit in recoding all assembly language procedures in PL/I. It must be noted that such a recoding has both a benefit and a cost: experiments suggest that while the number of source lines typically shrinks by slightly more than a factor of two, the number of generated machine instructions seems to increase by somewhat more than a factor of two, thus having some negative effect on the performance of the system [Huber, 1976].

The size impact of our studies is easiest to assess for four projects that were carried through to a trial implementation. Three of these had as their goal the outright removal from the kernel of the system of a certain body of code whose function we consider to be noncritical. Clearly, the impact of these modifications on the kernel size is the most dramatic and demonstrable. The extraction of the dynamic linker from the kernel [Janson, 1974] had the effect of removing 5% of the object code. More interestingly, it only removed 2 1/2% of the entry points inside the kernel, implying that most of the modules were fairly large; but it eliminated 11% of the entry points from the user domain into the kernel. In other words, removing this code from ring zero had a very strong effect in reducing the complexity of the interface that the user sees to the kernel. This should not be surprising, since we claim that the code did not belong in the kernel at all, and was in fact performing a user function. The project to remove some of the name management mechanism from the kernel [Bratt, 1975] did not have quite such a dramatic effect: it reduced the size of the kernel only by 2 1/2%. The latter project was dramatic chiefly in the reduction by a factor of four in

the total size of the code that implemented the algorithm once the algorithm was removed from the kernel. This was a case in which the complexity of the algorithm itself was due largely to the fact that it was inadvertently placed inside the kernel. Another project that had dramatic impact on the size of the kernel was an investigation of the Answering Service [Montgomery, 1976], the programs mentioned above that manage logins and accounting. Of the 10,000 lines of source code, it was shown that fewer than 1,000 of them need be included in the kernel.

The fourth study actually implemented, the redesign of the memory management algorithm [Huber, 1976], did not have as its goal the extraction of code from the kernel, but rather the restructuring of code in the kernel using parallel process, for the sake of clarity. The main size impact of this project came from recoding certain assembly language modules in PL/I, which had the impact reported above.

In terms of reducing the actual bulk of the kernel code, the most dramatic impact may come from a project that is only now being completed, and whose impact can therefore only be estimated. This project has to do with removal from the kernel of much of the code having to do with connection of the system to multiplexed networks [Ciccarelli, 1977]. Two multiplexed communication streams are attached to the Multics system: the ARPANET, and the local front end processor with all its attached terminals. At the start of the project, approximately 7,000 lines of PL/I were dedicated to handling these multiplexed lines, about 20% of ring zero. If a third network were to be connected to Multics, the original strategy would require that yet a third handler be added to this system. In other words, the bulk of the network control code would grow linearly with the number of networks attached. We are

now completing a project whose goal is to demonstrate that almost all of the network control software can be removed from the kernel into the user domain, and that much of the software that remains in the kernel to perform the actual demultiplexing of this stream can be, to a significant extent, constructed in a fashion independent of the particular network. Thus, the bulk of the kernel is much reduced, and only grows slightly as new networks are attached. While the results in this area are not yet demonstrable by a complete implementation, we estimate that this 7,000 lines of code in the kernel may shrink to less than 1,000, a reduction of 17% of the supervisor.

Another project whose size impact can only be estimated is the redesign of the system initialization mechanism, which proposed that certain parts of initialization be done in a user process environment in a previous system incarnation. We estimate that the removal of this code will shrink the kernel by 2,000 lines of PL/I.

It is useful to assess the combined effect of all the changes discussed above. The table below summarizes the various results.

| Kernel Size, Start of Project | | Reductions | |
|---|---|---|---|
| 44K | ring 0 | Linker | 2K |
| 10K | Answering Service | Name Manager | 1K |
| 54K | TOTAL | Answering Service | 9K |
| | | Network I/O | 6K |
| | | Initialization | 2K |
| | | Exclusive use of PL/I | 8K |
| | | TOTAL | 28K |

As this accounting indicates, the combined effect of our various projects could be to cut the size of the kernel roughly in half. At the start of the project, we had hoped that our impact on the bulk of the kernel could be somewhat greater than it was. Our optimism was, to a significant extent, based on the hope that projects such as the redesign of the memory manager would yield a simpler and thus smaller algorithm. In fact, the result was somewhat more subtle than this; the algorithm did get simpler, but not by outright elimination of pieces of code. Rather, the effect was elimination of paths between pieces of code. Operations originally in the kernel continue to be needed, but are executed under circumstances more constrained and better understood. Thus, the effect on absolute size is less than hoped, but the effect on complexity, although more difficult to gauge, is considerable.

Another area of interest is what might be the impact of specializing a Multics to be just a network-connected file storage system, with no general-purpose user programming permitted. Interestingly, many of the functions that one might expect to see deleted have already been removed from the kernel. Our best estimate is that such specialization might reduce the kernel size by at most another 15 to 25%.

Impact of redesign on performance

The effect of these projects on the performance of the system must be assessed. Our goal was not to achieve a performance improvement, but a significant performance degradation would be a cause for concern. In fact, the conclusion reached by most of the studies is that the performance of the system was not significantly affected by the proposed changes. While the dynamic linker ran somewhat slower when removed from the kernel, the causes

were well understood and curable. The name space manager ran somewhat faster. The revised Answering Service, in its preliminary implementation, ran about 3% slower.

The more interesting performance questions arise in connection with modules which, rather than being moved wholesale, were redesigned for clarity while remaining in the kernel. The two most interesting examples of this sort of modification are the new memory management and process management software. The process management software is interesting because the new design included a two-level process scheduler, a structure which in the past has not yielded good system performance although no one to our knowledge has been willing to claim such a failure in print. Unfortunately, the trial implementation that we are doing to explore this scheduler performance is not yet done. We have implemented and studied the bottom layer of the scheduler, and are confident that the combination of the layers will have a performance about the same as the current system. However, this claim is only speculative.

The performance of the memory management software was studied in detail. The new design was somewhat slower, for two important reasons. First, parts were recoded in PL/I from assembly language, which seemed to cost a factor of two in the speed of the code. Second, the new version of the memory manager used two dedicated processes to perform part of its function, while the original design ran all functions whenever a user took a page fault. This use of processes required memory management software to call the process management software, which added a small but unavoidable cost. On the other hand, the use of processes allowed part of the function to run at a low priority, when the processor might otherwise have been idle. This lower priority represents a performance improvement of uncertain magnitude. All

together, the performance impact of the new design would be negative, but not significant unless the system were cramped for memory and thrashing.

## Conclusion

The primary conclusion of this project is that the kernel of a general-purpose operating system (or of a specialized file-management system) can be made significantly simpler by imposing first a clear criterion as to what should be in it, and second a design discipline based on type extension. This simplification does not appear to lead to gross diseconomies, although minor performance problems do appear.

On the other hand, compared with kernel designs that have been proposed for less ambitious functions [Lipner, 1974] the kernel of a general-purpose system seems still to be a large program--30,000 lines of source code in this case study. And it is not apparent that specialization of the system to be just a file storage and management facility would make a very big reduction in this number--maybe 20%.

It is also apparent that minor adjustments of the underlying hardware architecture can make a significant difference in operating system complexity, and similarly that minor variations in the semantics of the user interface can make major differences in the complexity of implementation of the kernel.

Another lesson for designers is that one cannot hope to develop a modular design without consideration of the complete set of desired functions. If one leaves out, for example, resource control or reliability strategies for later addition, the chances are great that this addition will disrupt the module boundaries or introduce undesired dependencies.

With these several conclusions in mind, and the objective of a certifiable design as the goal, a designer of a new system should be able to create a design whose implementation can actually be reviewed for integrity, and used with confidence.

## Publications of the Kernel Design Project

### A. External Publications

Saltzer, J.H., "Protection and the Control of Information Sharing in Multics," Comm. ACM 17, 7 (July, 1974), pp. 388-402.

Saltzer, J.H., "Ongoing Research and Development on Information Protection," ACM Operating Systems Review 8, 3 (July, 1974), pp. 8-24.

Schroeder, M.D., "Engineering a Security Kernel for Multics," Proceedings of 5th Symposium on Operating Systems Principles, ACM Operating Systems Review 9, 5 (November, 1975), pp. 25-32.

Janson, P.A., "Dynamic Linking and Environment Initialization in a Multi-Domain Process," Proceedings of 5th Symposium on Operating Systems Principles, ACM Operating Systems Review 9, 5 (November, 1975), pp. 43-50.

### B. External Publications in Preparation

Gifford, D., "Hardware Estimation of a Process' Primary Memory Requirements, Version II," submitted to CACM.

Schroeder, M.D., Clark, D.D., and Saltzer, J.H., "The Multics Kernel Design Project," submitted to Sixth ACM Symposium on Operating Systems Principles.

Reed, D.P., and Kanodia, R.J., "Synchronization with Eventcounts and Sequencers," submitted to Sixth ACM Symposium on Operating Systems Principles.

Kanodia, R.J., and Reed, D.P., "Synchronization in Distributed Systems," in preparation.

Janson, P.A., "Using Type-Extension to Organize Virtual-Memory Mechanisms," in preparation.

### C. Theses and Technical Reports

Janson, P.A., "Removing the Dynamic Linker from the Security Kernel of a Computing Utility," S.M. thesis, Massachusetts Institute of Technology,

Department of Electrical Engineering and Computer Science, June, 1974, also Project MAC Technical Report TR-132.

Bratt, R., "Minimizing the Naming Facilities Requiring Protection in a Computer Utility," S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, July, 1975, also Project MAC Technical Report TR-156.

Gifford, D., "Hardware Estimation of a Process' Primary Memory Requirements," S.B. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May, 1976, also Laboratory for Computer Science Technical Memorandum TM-81.

Huber, A., "A Multi-process Design of a Paging System," S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May, 1976, also Laboratory for Computer Science Technical Report TR-171.

Montgomery, W., "A Secure and Flexible Model of Process Initiation for a Computer Utility," S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June, 1976, also Laboratory for Computer Science Technical Report TR-163.

Reed, D., "Process Multiplexing in a Layered Operating System," S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June, 1976, also Laboratory for Computer Science Technical Report TR-164.

Janson, P., "Using Type Extension to Organize Virtual Memory Mechanisms," Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, August, 1976, also Laboratory for Computer Science Technical Report TR-167.

Hunt, D., "A Case Study of Intermodule Dependencies in a Virtual Memory Subsystem," E.E. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, December, 1976, also Laboratory for Computer Science Technical Report TR-174.

Goldberg, H., "Protecting User Environments," S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, January, 1977, also Laboratory for Computer Science Technical Report TR-175.

D. Theses and Technical Reports in Preparation

Luniewski, A., "A Certifiable System Initialization Mechanism," S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, thesis completed January, 1977, Laboratory for Computer Science Technical Report in preparation.

Mason, D., "A Layered Virtual Memory Manager," S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and

Computer Science, expected date of completion, April, 1977.

Ciccarelli, E., "Multiplexed Communication for Secure Operating Systems," S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, expected date of completion, June, 1977.

Feiertag, R., "A Methodology for Designing Certifiably Secure Computer Systems," Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, expected date of completion, June, 1977.

E. Annual Reports

M.I.T. Project MAC Annual Report XI, 1973-74, pp. 155-183.

M.I.T. Project MAC Annual Report XII, 1974-75, (in preparation)

M.I.T. Laboratory for Computer Science Annual Report, 1975-76, (in preparation)

M.I.T. Laboratory for Computer Science Annual Report, 1976-77, (in preparation)

References

Bell, D., and LaPadula, L., "Secure Computer Systems," Air Force Elec. Syst. Div. Report ESD-TR-73-278, Vols. I, II, and III, November, 1973.

Bredt, T., and Saxena, A., "A Structured Specification of a Hierarchical Operating System," ACM Proc. Int. Conf. on Reliable Software 10, 6 (June, 1975), pp. 310-318.

Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Trans. on Software Engineering SE-1, 2 (June, 1975), pp. 199-207.

Lampson, B., "A Note on the Confinement Problem," Comm. ACM 16, 10 (October, 1973), pp. 613-615.

Lampson, B.W., et al., "Report on the Programming Language EUCLID," Defense Advanced Research Projects Agency Information Processing Techniques Office Report, August, 1976.

Lipner, S., Chm., "A Panel Session--Security Kernels," AFIPS Conf. Proc. 43, NCC 1974, pp. 973-980.

Neumann, P., et al., "A Provably Secure Operating System," Final Report on SRI Project 2581, Stanford Research Institute, 1975.

Ritchie, D.M., and Thompson, K., "The UNIX time-sharing system," CACM 17, 7 (July, 1974), pp. 365-375.