MIT/LCS/TR-196

FINAL REPORT OF THE MULTICS KERNEL DESIGN PROJECT

Schroeder, Clark, Saltzer & Wells

This blank page was inserted to preserve pagination.

MIT/LCS/TR-196

FINAL REPORT OF THE MULTICS KERNEL DESIGN PROJECT

by

M.D. Schroeder*
D.D. Clark
J.H. Saltzer
D.H. Wells

June 30, 1977

This research 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.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

This empty page was substituted for a blank page in the original document.

FINAL REPORT OF THE MULTICS KERNEL DESIGN PROJECT

FOREWORD

This report summarizes a three-year project to develop a simpler version of the supervisor of the Multics operating system, so that auditing for security certification might be feasible. The report is in four sections:

I. A summary of the highlights of the project, together with a complete list of published papers and technical reports of the project.

II. A short description of every individual task undertaken as part of the project.

III. An estimate of the potential impact on the size of the Multics Kernel if every idea suggested for simplification were implemented.

IV. Conclusions and recommendations.

Together, these four sections provide a system designer with a high-level description and many pointers to more detailed analyses of issues involved in securing a large-scale, general purpose computer system.

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

This empty page was substituted for a blank page in the original document.

PART I: THE MULTICS KERNEL DESIGN PROJECT

by

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

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 into 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.

This empty page was substituted for a blank page in the original document.

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 [Saltzer, CACM, 1974]. 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 seemed 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. These aspects raised, in turn, three questions: 1) could auditability really be achieved? 2) is a formally modelable security function usable?, and 3) what happens to the system's performance? To answer these questions, the overall project was broken into several small components that allowed orderly experimentation and took 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 confinement and control of information flow among the levels and compartments [Bell and LaPadula, 1973]. The first step in this project (the box numbered 1 in figure 1) was to take the standard Multics system, and systematically add to it, so far as possible, the 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

*Actually, a predecessor of the MITRE model devised by a team at Case Western Reserve [Walter, 1974] was used for this step. The later-developed MITRE model is consistent with that earlier model, and all recent work has used the newer model.

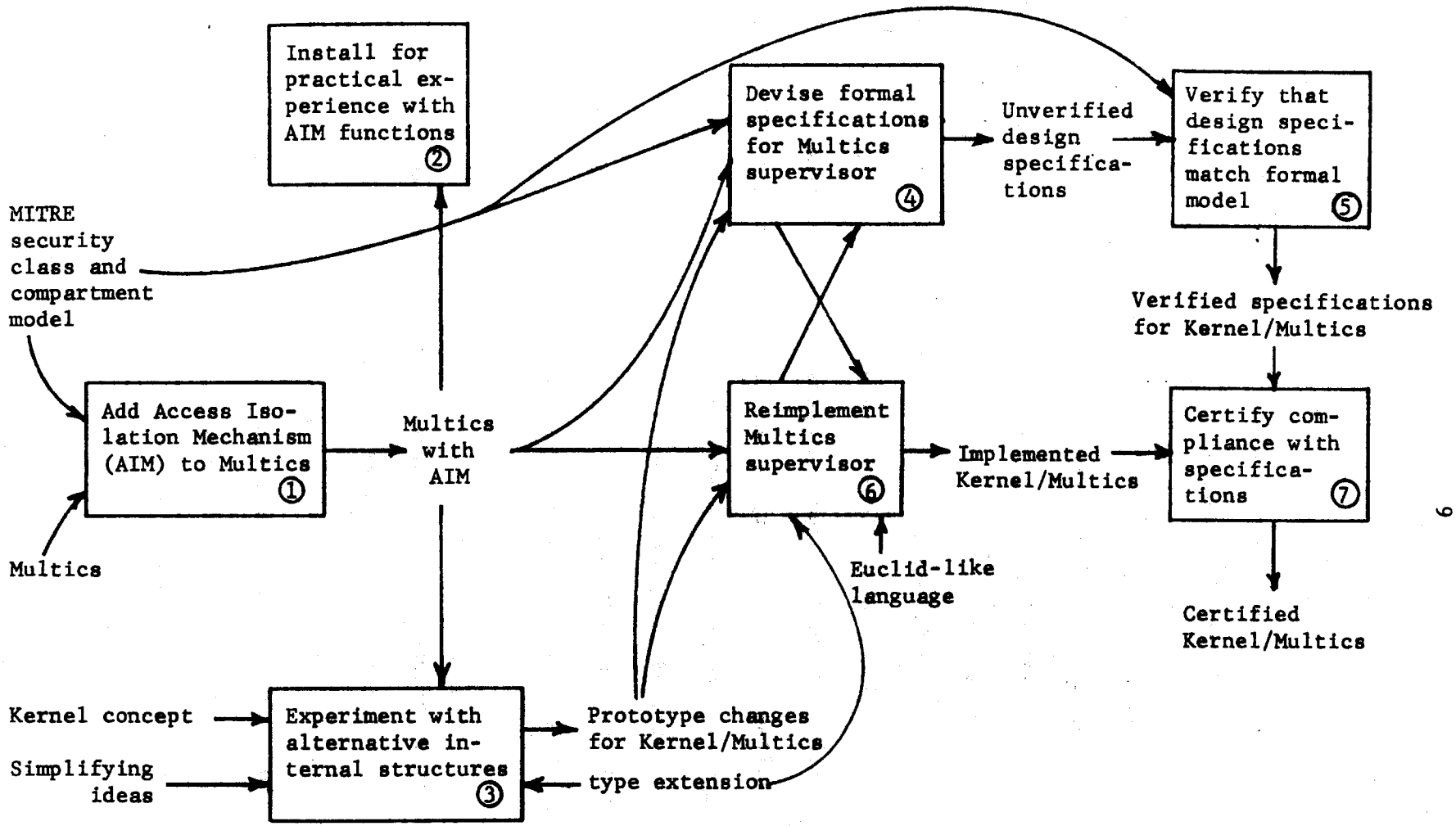


Figure 1 -- Plan for developing a certifiable security kernel for Multics

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 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 and with its impact on performance. A series of prototype implementations were undertaken to discover ways of accomplishing the same functions with simpler and more systematic operating system structures, while keeping the discipline of the security kernel concept [Schell, 1973] in mind (box 3 of the figure). And two groups of analysts began to develop successively more detailed examples of formal specifications for the design 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 paths to be undertaken in parallel: first, the formal specifications for the design of a Multics kernel (box 4) must be completed and they must be verified (box 5) as matching the requirements of the MITRE security model. The second parallel path (box 6) involves a reimplementation of the central supervisor of Multics, with two differences from the present implementation: those prototype simplifications that were successful will be incorporated, and the form of the design and 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 design discipline, and by using a programming language that is designed to support verification, such as EUCLID [Lampson et al., 1977] or a constrained subset of PL/I.

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 design specifications that have been verified to match the MITRE security model. The final step, box 7 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 techniques 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

---

* 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 Air Force. 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, 3, and most of 4 of figure 1 completed, the Air Force suspended work on the project.

** The "kernel" that is referred to here is defined as all programs that implement or affect access control of any kind, discretionary or non-discretionary. Therefore, a substantially larger body of programs is involved than in a security kernel that implements only non-discretionary controls, such as that described by MITRE [Lipner, 1974].

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 separation afforded by independent processes in distinct address spaces communicating at arms length to implement protected functions, and 3) using more systematic program structuring techniques for implementing the remaining 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. 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. This way of thinking about modules has been developed by the programming languages community over the last several years [Liskov and Zilles, 1975].

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

partially ordered set, allows system correctness to be established one module at a time. This argument was first exploited in the THE system [Dijkstra, 1968] and more recently in the system design by SRI [Neumann, 1977].

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 Storage Dependencies

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

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. While some of these dependencies may not fit naturally into this classification, proper classification is of no concern, 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.

Getting 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 one process causes page control to invoke process control, which in turn invokes segment control to load the state of another process 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, several processes simultaneously may try to cause page control to alter the state of the same page. A global lock regulates such conflicts. Unfortunately, the hardware imposes a short time window between a missing page exception and the setting of the lock by page control during which time some other process may alter the address translation tables. Once the lock is

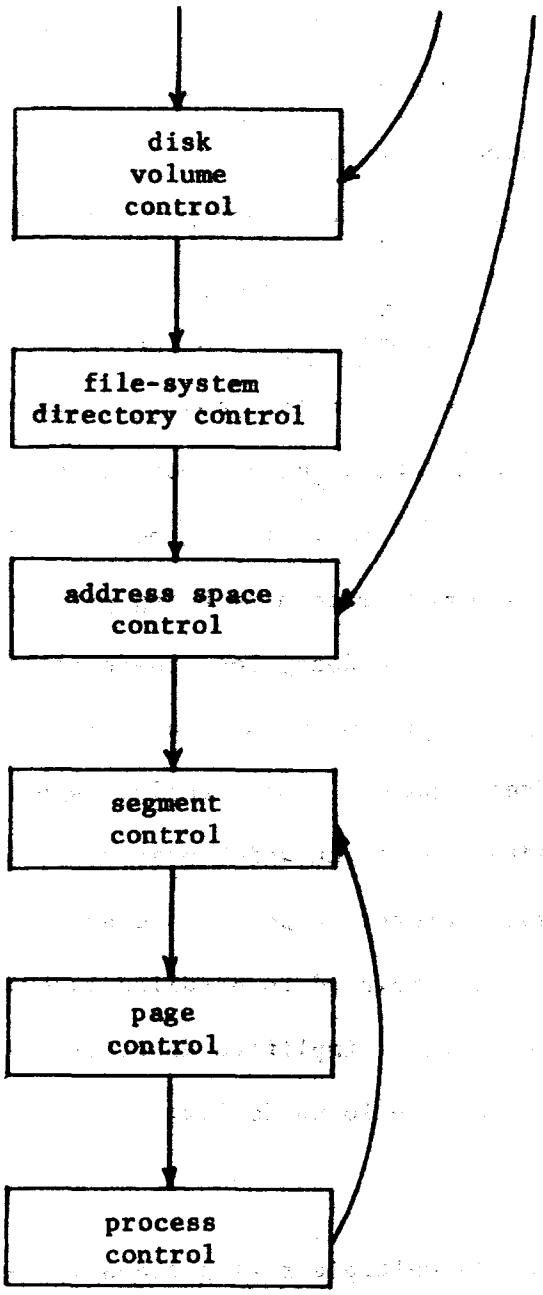


Figure 2 -- Superficial Dependency Structure
in Multics.

captured, page control must interpretively retranslate the virtual address that caused the exception to see if the same exception is still encountered. This interpretive retranslation requires page control to know the format of and to depend upon the correctness of the address translation tables maintained by segment control and address space control.

b. Quota Enforcement

Arbitrary directories in the file system hierarchy 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 enlarged, 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 enlarge 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 and its quota directory. This implementation of quotas and storage usage records makes page control depend on segment 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. Enlarging 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. If a full disk pack exception is detected when enlarging a segment, 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 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.

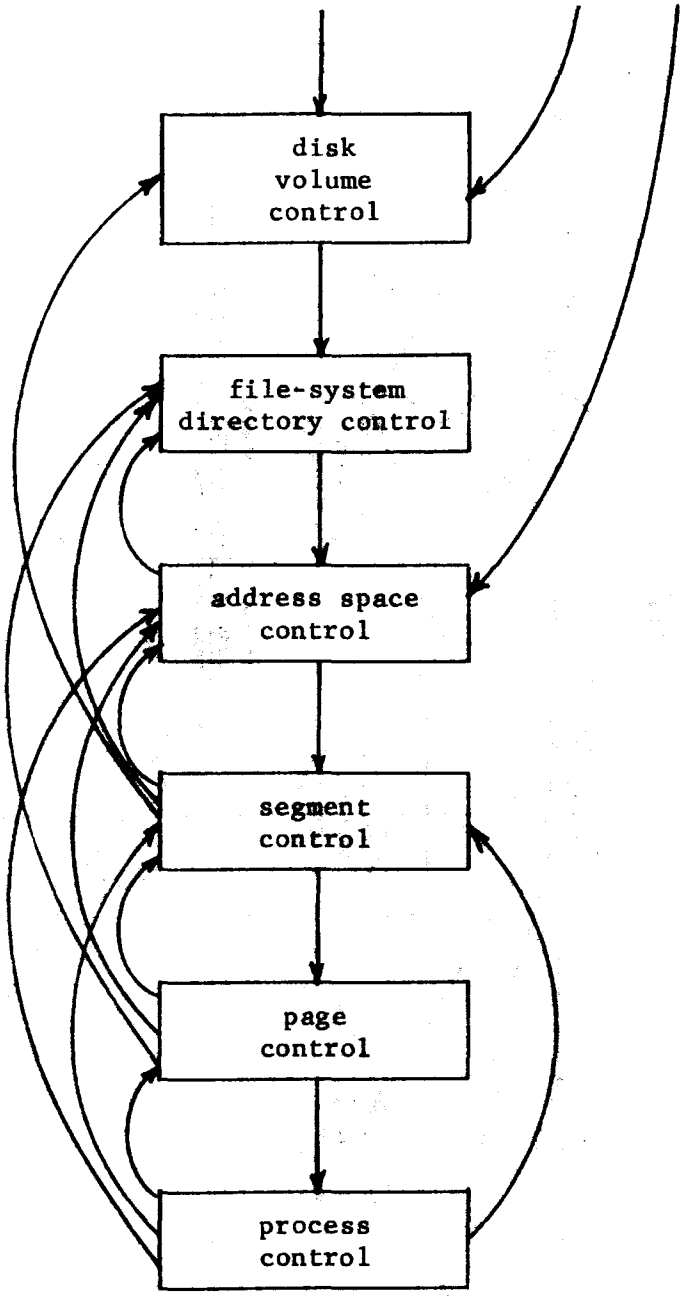
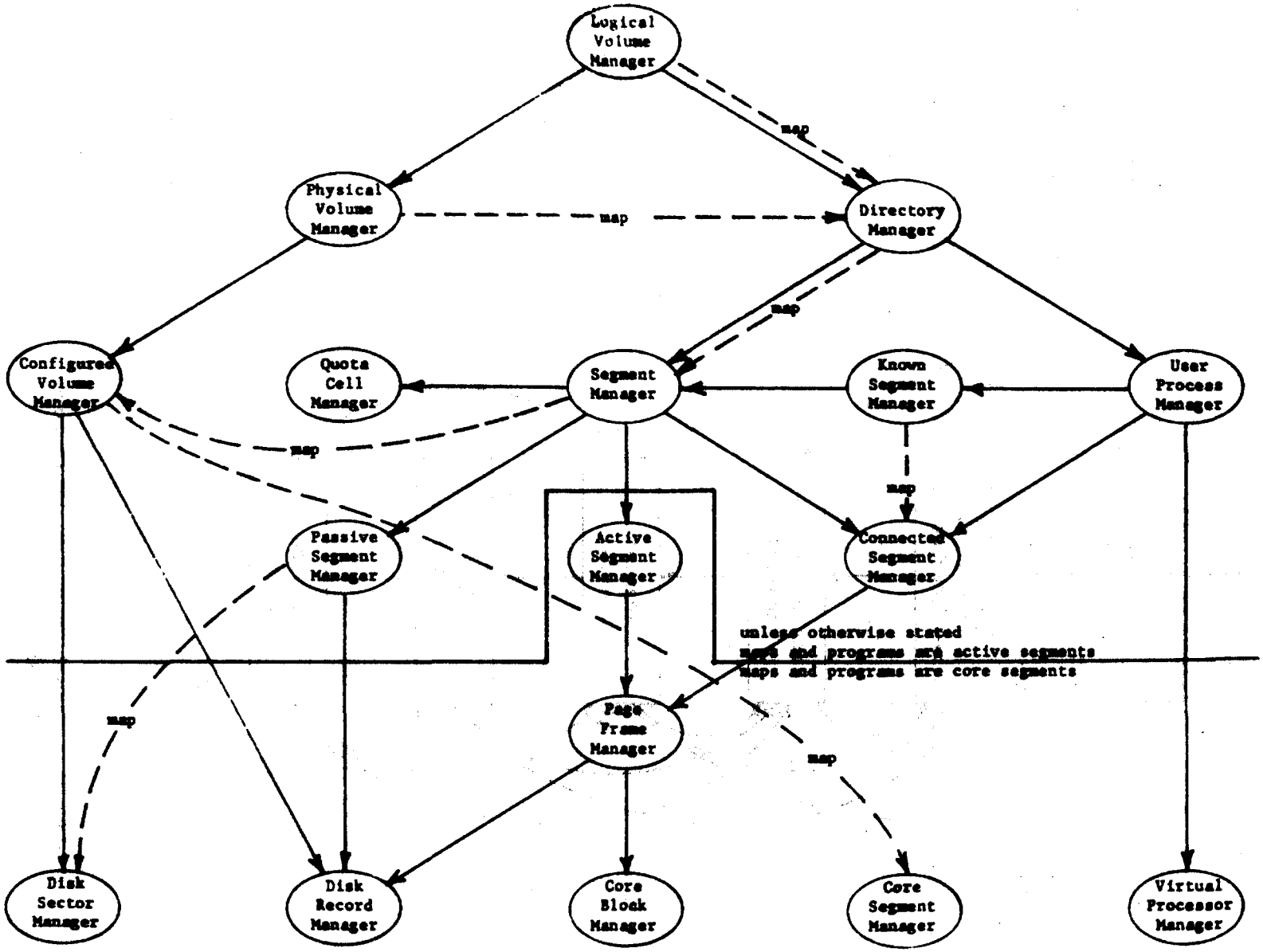


Figure 3 -- Actual Dependency Structure
in Multics



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

Figure 4 -- Restructured file system, with dependency loops removed.

The loop between the processor multiplexing facilities and the virtual memory mechanism originates from the goal of providing a variable number of processes. Brinch Hansen has argued that considerable simplification of implementation follows a decision to implement a fixed number of processes [Brinch Hansen, 1975]. On the other hand, if one tried to open the dependency loop between process implementation and virtual memory implementation, every process state would have to be resident in the fastest, most expensive memory medium. If the number of processes were 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. Process control is divided 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, and all the simplifying advantages suggested by Brinch Hansen occur. 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 is 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.

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 left a key complicating factor as an exercise for the implementor: when a low-level virtual processor discovers an event that it must signal to a user-level process, it must somehow change the state of the

user-level receiving process. But that 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]. Use of a two-level process implementation in the Multics kernel is worked out in sufficient detail that we are confident that 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 storage 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 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 (or reconfigured) and thereafter the only operations on them available to higher levels 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 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

* 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.

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 solutions to the dependency loops associated with quotas and full disk packs illustrate two alternative ways of reporting exceptional conditions without creating dependencies. A problem common to both situations is that software in some module may discover 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. There is a basic strategy that can break these dependency loops: to transfer 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. This strategy can be carried out either by a hardware interrupt or by a carefully planned software signalling mechanism. Both approaches are illustrated below.

In the case of quota enforcement and recording disk usage, recall that an attempt to enlarge a segment, and thus the need 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 rather than missing-page exceptions. 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 of 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.

The loop associated with full disk packs is broken by the use of a software mechanism for signalling exceptions upward in the dependency structure. 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 only rarely 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 a chain of calls down through the dependency 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. The segment manager finishes all its work and prepares to restart the user process, but rather than restarting it passes control directly to the directory manager as though an exception had just occurred*. 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 restore conditions to the point of the

* The trick of passing an exception to another program better equipped to handle it by making things look as if that other program had been called originally is an old one, used in many systems. The interest here is that it can be used to break dependency loops.

original exception and the user process then references the segment again. At this point any other process referencing the segment will be reconnected via the standard machinery for handling missing segment exceptions.

This completes the discussion of the dependency problems found in Multics and the methods used to deal with them. 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 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, has already been discussed in the section on loop dependencies. The dynamic nature of quota directories implies at every quota exception a new search for the relevant quota cell by following a linked chain of directory entries in the active segment table. In order to maintain this linked chain segment control must be careful never to deactivate a segment that is a directory if inferior segments in the hierarchy are active. Thus segment control is constrained to manage the active segment table to track the shape of the directory hierarchy defined by directory control. In this case, a slight change of semantics seemed worthwhile: restrict the dynamic designation or undesignation of directories and quota directories to those directories that have 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 enlarged. 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. As a result, the deactivation of segments by the active segment manager no longer is constrained by the shape of the directory hierarchy.

For another example of complicating semantics, a combination of two simple access control ideas 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. The first simple idea is that directories should 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. The second simple idea, to make the semantics of access control as simple as possible, is the rule 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 semantics 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

---

* Note that this set of issues deals entirely with the semantics of discretionary control. In a kernel design that focused exclusively on non-discretionary control, the interaction between access control and name resolution would be relegated to applications program implementation.

secondary (disk) memory storage charges. The user interface specifies 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. As described earlier, 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 in primary memory. 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, 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 file storage and management. We believe that they would.

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 be based primarily on examination of the machine instructions themselves. The most useful and consistent measure of the kernel size seems to be the number of source lines, independent of the language being used, and this is the measure we shall 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 44,000 lines of source code 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, which regulates attempts to log in to the system, including authenticating

passwords, and manages system accounting. These programs contained about 10,000 lines of source code. It is clear that the non-ring-zero programs contribute significant bulk to the kernel of the system. As a starting point, then, we consider the kernel to have consisted of 54,000 lines of source code.

As mentioned above, some of the kernel is coded in assembly language rather than PL/I. Because of this, 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 2000 lines of source code, about 4%. More interestingly, it only removed 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 the kernel 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 1000 lines. 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 processes, 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, another 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 12% of the kernel. 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.

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 [Luniewski, 1977]. We estimate that the removal of this code will shrink the kernel by 2,000 lines.

It is useful to assess the combined effect of all the changes discussed above. Table one summarizes the various results. 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 there, 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.

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

Table 1

Summary of Kernel Size Reductions

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%, mostly by allowing simpler algorithms to manage the more constrained environment.

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 was intended to explore this scheduler performance was not completed. 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 in the process of the user that 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.

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--the kernel concept--,and second a design discipline based on type extension. The kernel concept seems to be a viable approach to security in large-scale systems as well as in the small-scale ones to which it has been previously applied.

On the other hand, compared with kernel designs that have been proposed to deal exclusively with non-discretionary control [Lipner, 1974] the kernel of a general-purpose system seems still to be a large program--26,000 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%. At the same time, there does not seem to be a significant performance loss arising from use of simpler, more modular designs. This observation reinforces observations made, as part of the larger project, that in production use Multics with AIM performs no

differently than Multics without AIM. Together, these observations lead to a very strong conclusion that a secure system need have no performance penalty.

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 conclusion 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's Primary Memory Requirements," to be published in Comm of ACM, September, 1977.

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

Reed, D.P., and Kanodia, R.J., "Synchronization with Eventcounts and Sequencers," to appear in the 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

The following are theses submitted to the Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, and are available as M.I.T. Laboratory for Computer Science Technical Reports.

Janson, P.A., "Removing the Dynamic Linker from the Security Kernel of a Computing Utility," S.M. thesis, June, 1974, Technical Report TR-132.

Bratt, R., "Minimizing the Naming Facilities Requiring Protection in a Computer Utility," S.M. thesis, July, 1975, Technical Report TR-156.

Gifford, D., "Hardware Estimation of a Process' Primary Memory Requirements," S.B. thesis, May, 1976, Technical Memorandum TM-81.

Huber, A., "A Multi-process Design of a Paging System," S.M. thesis, May, 1976, Technical Report TR-171.

Montgomery, W., "A Secure and Flexible Model of Process Initiation for a Computer Utility," S.M. thesis, June, 1976, Technical Report TR-163.

Reed, D., "Process Multiplexing in a Layered Operating System," S.M. thesis, June, 1976, Technical Report TR-164.

Janson, P., "Using Type Extension to Organize Virtual Memory Mechanisms," Ph.D. thesis, August, 1976, Technical Report TR-167.

Hunt, D., "A Case Study of Intermodule Dependencies in a Virtual Memory Subsystem," E.E. thesis, December, 1976, Technical Report TR-174.

Goldberg, H., "Protecting User Environments," S.M. thesis, January, 1977, Technical Report TR-175.

Luniewski, A., "A Certifiable System Initialization Mechanism," S.M. thesis, January, 1977, Technical Report TR-180.

Mason, D., "A Layered Virtual Memory Manager," S.M. thesis, June, 1977, Technical Report TR-177.

Clark, D., editor, "Ancillary Reports of the Kernel Design Project," June 30, 1977, Technical Memorandum TM-87.

D. Theses and Technical Reports in Preparation

Ciccarelli, E., "Multiplexed Communication for Secure Operating Systems," S.M. thesis, expected date of completion, September, 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, pp. 89-104.

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)

Additional 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.

Dijkstra, E.W., "The Structure of the THE-Multiprogramming System," Comm. ACM 11, 5 (May, 1968) pp. 341-346.

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," SIGPLAN Notices 12, 2 (February, 1977) pp. 1-79.

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

Liskov, B.H., and Zilles, S., "Specification Techniques for Data Abstraction," IEEE Trans. Software Engineering SE-1, 1, (1975) pp. 7-19.

Neumann, P., et al., "A Provably Secure Operating System: the System, its Applications, and Proofs," Final Report on SRI Project 4332, Stanford Research Institute, February, 1977.

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

Schell, R., "Notes on an Approach for Design of Secure Military ADP Systems," in Prelimary Notes on the Design of Secure Military Computer Systems, United States Air Force Electronic Systems Division MCI-73-1, January, 1973, pp. 1-1 through 1-5.

Walter, K.G., et al., "Primitive Models for Computer Security," United States Air Force Electronic Systems Division Technical Report ESD-TR-74-117, January 23, 1974.

PART II: KERNEL DESIGN PROJECT TASK REPORT

by

David D. Clark

Introduction

The kernel design project was composed of twenty-two individual tasks. This section of the final report discusses each of the tasks initiated during the course of the project.

Many of the tasks described here have been documented in greater detail. In most cases this documentation is in the form of a Technical Memo (TM) or Technical Report (TR) of the Laboratory for Computer Science. A complete bibliography of the project appears in Part I of the report.

I. Studies of Formalisms for System Specification

At the beginning of this project, we invested a certain amount of effort in exploring known techniques for expressing the specification of operating systems. While we did not intend, as part of our research, to construct a formal specification for the Multics operating system, it was important for us to understand enough about the construction of specifications to see how our work would relate to this task. We experimented with three different specification languages: the Vienna Definition Language, a stylized English, and a special language developed here and locally known as GSPL, a PL/1-like language with data structures based on LISP. In an attempt to discover the relevance of structured programming to our project, structured representations of two parts of the system, page control and traffic control, were developed. These preliminary experimentations proved very valuable in developing the group insight. The structured representation of page control in GSPL forms an appendix to technical report TR-127 by B. Greenberg.

II. Analysis of Original System

Before we could begin to perform any organized rearrangement of the kernel of Multics, it was necessary to have a clear idea of what was contained in the kernel of the system as it existed at the beginning of our project. To this end, the programs that constituted the supervisor of the existing system were analyzed in several ways. First, we gathered together the functional specification for every entry point into the supervisor. The resulting notebook constituted a first cut at a functional specification of the Multics kernel. Second, all of the segments that constituted this supervisor of the system were categorized by function and by source language. The results of

this preliminary assessment, and a comparison with the system of today, are summarized in the earlier portion of this final report. The preliminary assessment is reported in "A Census of Ring 0" by V. Voydock, reprinted in TM-87.

III. Formulation of Criteria for Inclusion of Modules within the Kernel

There are a variety of forces that have caused modules to be moved into the Multics supervisor. Some of these modules are obviously related to maintenance of system security, others have something to do with system security, but might be removable at least in part, and others exist in the supervisor for reasons such as efficiency or convenience, and are not related to maintenance of system security in any way. We believed that the size of the supervisor could be markedly reduced by dissecting a large number of system modules and removing them, either partially or wholly, from the supervisor. Before we could begin such a removal process, however, it was necessary to determine exactly what criteria we would use to justify the inclusion or exclusion of a module from the kernel. We began by studying a number of specific parts of the current system and identifying the trade-offs related to removing these particular parts out of the kernel. One study in particular was performed of page control. We identified three levels of security with which we might be concerned, protection of information from direct release or modification, denial of service, and confinement of user computation to protect against leakage by means of a "trojan horse" attack. In general, we adopted the principle that protection against confinement was not easily achievable in today's environment, and that protection against denial of service was achievable and important, but that denial of service was less important than direct unauthorized release or modification of data. A discussion of these kernel levels is contained in TR-163 by W. Montgomery.

IV. Analysis of Flaws in the Multics System

In an attempt to understand the sorts of problem that lead to potential violations of security, our group periodically collected and documented every known way to penetrate the Multics system. While the list of uncorrected bugs was not circulated, we periodically issued a report which analyzed bugs after a repair had been installed in the system. These analyses are of a very pragmatic nature, but yield considerable insight into the sort of problem that must be solved in practice if a secure system is to exist. These reports were reprinted in TM-87.

V. Performance Benchmark for the Multics System

One of our concerns in this project was that the performance of the system should not be significantly degraded by the modifications that we proposed. We had anticipated using the standard Multics benchmark developed at the MIT Information Processing Center to evaluate our modified versions of the system, but we discovered that this benchmark was too time consuming and not sufficiently precise for our purposes. For this reason we invested some effort in producing a variant of this benchmark that ran more quickly than the standard version and whose results were more repeatable. We produced a version of the benchmark that started and stopped the calibration tasks in such a way that the resulting running conditions were much more repeatable than in the standard benchmark. This modified benchmark was used to produce the performance results reported earlier in this report.

We also invested some effort in designing a version of the benchmark that provided the test load on the Multics system using interactive processes logged in over the ARPANET, as opposed to the absentee jobs used by the standard benchmark. The advantage of interactive processes is that they

exercise the system in a fashion more similar to the way the system is actually used. This latter project was never completed. It appeared that the need for an evaluator of this complexity and accuracy was not required, since the majority of the engineering studies that we performed were not carried through to an implementation that was sufficiently tuned to yield more than very rough performance information.

We performed two other small projects related to performance monitoring and evaluation. One project was experimental observation of various classes of users on the system, in order to develop an empirical model of the arrival pattern of user commands. This work is reported in an undergraduate thesis by H. Rodriguez, entitled "Measuring User Characteristics on the Multics System". We also imported and made operational a performance monitoring package called "aware" originally developed by the Ford Motor Company.

VI. Removal of the Dynamic Linker from the Kernel

Our preliminary analysis of the Multics kernel indicated that a significant volume of the kernel consisted of programs that did not need to be in the kernel for reasons of security, but were there for reasons of efficiency or tradition. It was important to determine whether or not it was practical to remove these modules bodily from the kernel. In most cases it was clear that some small percentage of the function did require supervisor privilege, and there was some fear that this residue would complicate the outright extraction of the remainder. The first such task which we undertook was the removal of the dynamic linker from the kernel. The dynamic linker, which translates at run time between symbolic names and segment numbers, was an obvious candidate for removal for four reasons. First, the linker did not implement any concept related to the protection of the system or needed to

support the protection mechanisms. Its function is entirely related to the execution of user written code. Second, in view of the function implemented by the linker, it seemed reasonable to suspect that the linker did not need any of the privilege granted to typical modules of the security kernel. Third, the linker was a very complex program. Even though its function was easy to describe, the details of its implementation required the use of intricate and sophisticated language constructs that made the reading and auditing of the program an almost impossible task. Finally, the linker, by its very nature, handles data directly accessible to the users of the system. Such data could contain, purposely or not, inconsistencies capable of causing the linker to malfunction or perform unexpected operations. It seemed much harder to verify the correct operation of a program when that program could be presented with an arbitrary input than to verify correct operation when a "correct" input was guaranteed. Very sophisticated machinery would be required to verify the consistency of user databases and thus insure proper operation of the linker. Inclusion of such machinery, if possible, would only increase the complexity of the linker. The alternative of removing the linker from the kernel would insure automatically that no malfunction of the linker would ever subvert the protection mechanism of the system.

Since this project was one of our earliest, the design was carried through to an implementation in order to increase our confidence that the techniques we were proposing in principle would work in practice. The completed implementation also allowed us to make some preliminary performance studies, since there was some concern that removal of functions from the kernel might significantly degrade the performance of the system. The conclusions drawn from this project were that the outright removal of certain

functions from the kernel was indeed feasible and practical, that no drastic performance degredation need be expected in practice, and that the flexibility of the system was in fact enhanced by this extraction, since the user now had the option of replacing the linker with an alternative program of his own choice. One useful byproduct of this study was the conclusion that kernel intervention is not required when control is being transfered between one user domain and another, even if those two domains are mutually untrusting. This is a most interesting conclusion, which was not at all obvious at the beginning of the project.

The results of this project are reported in detail in technical report TR-132, and in "Dynamic Linking and Environment Initialization in a Multi-Domain Process", Proceeding of 5th Symposium on Operating Systems Principles, ACM Operating Systems Review 9, November 1975, both by P. Janson.

VII. Minimizing the Naming Facilities Requiring Protection

This project involved identifying another component of the existing Multics kernel that could be removed bodily into the user environment. Multics provides a very sophisticated naming environment that users may use to keep track of their files. One set of names available to the user, file system names, are global in scope and can be used by any user to identify a shared file. Since these names are shared among users, it is not obvious how their management could be removed from the kernel. However, there are other sorts of names, reference names, private to each user, which provide an efficient way of naming a file already identified using a file system name. Since the management of reference names is private to each user, it seemed reasonable to remove their management from the kernel.

Removing the reference name manager from the kernel required that a kernel data base, the known segment table, be split into a private and a common part. As part of this change, the interpretation of path names was also removed from the kernel. As discussed in the first part of this report, this required that the supervisor learn to lie convincingly on occasion about the existence of certain file system directories.

This project was also carried through to an implementation, primarily because we anticipated demonstrating a performance improvement, and a drastic reduction in the complexity of the algorithm once we eliminated the constraints imposed on the algorithm by the necessity of its shared operation in the kernel. The result was a reduction by a factor of five in the kernel code required to manage the address space of a process, and an improvement in performance. A new and simpler kernel interface was an additional by-product.

The results of this research are represented in technical report TR-156 by R. Bratt.

VIII. Removal of the Global Naming Hierarchy from the Kernel

The previous task description discussed the existence of a global naming environment, the Multics file system. Since this naming environment is shared among all the users, it was not at all obvious that this name management mechanism could be removed from the kernel. However, it appeared that the file system could at least be partitioned into two parts, a single-layer catalog of segments, indexed by unique id, and a higher level name management mechanism which performed no function except the mapping between user provided names and unique id's. If such a division could be performed, then it would be possible to imagine removing this higher level from the kernel, and

providing a different copy of this management package for users in each different security compartment. While this would segregate the users into disjoint classes that would be incapable of refering to each others files, such a segregation might be acceptable in many applications. Even if it were not possible to remove this name management algorithm from the kernel, the partitioning of the algorithm into two components would presumably increase the modularity of the system, which would enhance the auditibility of the kernel. This project was initiated, but not completed. It was clear that this was a very major upheaval to the functionality of Multics, in addition to being a major upheaval to the structure of the existing code. We felt that for our purposes the effort required to perform this surgery would not be appropriate, given the requirement that we conform to the current Multics specification. In a new system that was being designed with the goals of auditibility in mind, we would strongly urge that this structure be considered, and if Multics were being completely redesigned, we think that it would be quite valuable to evaluate this structure for inclusion.

IX. Study of Multics System Initialization

If one is to certify that a system works correctly, one must begin by verifying the "initial state" of that system. For this reason, it was very important to understand how the Multics system initialized itself. The original initialization procedure was relatively unstructured in the sense that we found it very difficult to understand how one might verify its operation. Essentially, initialization proceeded in a number of very small incremental steps, each of which augmented the environment of the programs which followed it. This meant that each program ran in a slightly different environment than its predecessor. It was characterizing this large number of

different environments which made verification of program correctness so difficult. The reason for this large number of incremental steps performed during every initialization is that each of these steps represents a point at which the system can be tailored to reflect the particular physical configuration of the hardware available at the moment. Thus, a single Multics tape containing the initialization programs could be generated that would bring up a running Multics on any configuration, in contrast to other systems that require the generation of a different tape specific to each particular configuration.

We proposed an alternative structure for Multics initialization that continued to achieve this goal, but which we considered to be much more amenable to verification. Our strategy divided initialization into two phases. In the first phase, a bit string that consituted a version of Multics capable of running on any configuration was loaded into memory. In order to do this, it was necessary to demonstrate that there was a minimal set of hardware and software which constituted a subset of every viable configuration. Once we had defined this minimal configuration, it was possible to generate a version of Multics that used just these resources. The generation of this minimal Multics was done not at the time the system was initialized, but at the time the tape was generated. Generating the minimal Multics at tape generation time makes validating the generation programs much simpler, since the programs can run on a full fledged Multics, rather than in the environment that they are attempting to create. The second phase of initialization consisted of a series of dynamic reconfigurations that modified the minimal Multics to conform to the particular available hardware and operating parameters at this site. Dynamic reconfiguration has always been an

essential part of Multics, and many of the reconfigurations required for this purpose already existed in this system. However, it was necessary to demonstrate that certain supervisor tables, such as the traffic control and segment management data bases, could be grown, and an implementation was performed to prove this particular claim. Although this initialization strategy was not completely implemented, we are very confident that it is easily amenable to validation, since it conforms in its structure to the principles of layering, which appear to be powerful principles in operating system structuring.

The results of this work are reported in technical report TR-180 by A. Luniewski.

X. Restructuring of Page Control

The Multics kernel is implemented as code distributed among all the processes in the system. That is, a user desiring a particular service of the supervisor executes the relevant supervisor code in his own process. There is an alternative structure, in which the supervisor is implemented as separate processes that communicate with the user using interprocess communication mechanisms. This alternative, in certain cases, has the advantage that it isolates as a sequential process an algorithm that by its nature wants to be sequential, but that had been forced to an unnatural structure by being executed, potentially in parallel, by several user processes. We were very anxious to explore the use of this strategy within the Multics kernel.

The part of the supervisor that we chose as a testbed for this experiment was the low level memory management, commonly called page control. When a user references a page not in main memory, the page must be fetched from

secondary storage into an empty location in main memory. In order to perform this move, it may be first necessary to create an empty space in main memory by removing some other page. This removal algorithm has traditionally been run at the time of a page fault, but there is no necessity that it be run then. Our belief was that the removal algorithm could be more sensibly structured as a separate process, running in parallel with user processes, with no function other than to identify and remove from main memory pages not recently used. By segregating this algorithm in a separate process, the user process is no longer concerned, at fault time, with such problems as queuing disk writes, and waiting for their completion. Rather, the users process performs a very simple operation: it requests an empty piece of main memory, abandoning the processor if necessary until one is available, and then performs a read operation from secondary storage into this location.

A redesign of page control also allowed us to explore the implications of recoding certain assembly language programs in PL/1. The page control algorithms had been coded in assembly language for efficiency, and we were anxious to find out exactly what the impact would be of using a higher level language. The redesigned page control was implemented, since we were interested in investigating the performance characteristics of the system and since we wanted to confirm, by actually running the system, that we had identified all interactions between the page control functions now isolated in separate processes, and the higher levels of the supervisor still running in user processes. In fact, these connections between the removal process and the higher levels of the supervisor turn out to be some of the stickiest problems associated with this version of page management. The problem is that higher level functions occasionally request that particular pages they specify

be removed from primary memory, and this explicit request from above does not fit neatly into the otherwise clean pattern of the removal algorithm. The alternative of having these explicit removal operations performed by the user process implies that more than one process can be removing pages from memory at the same time, which in turn implies that the data bases describing the contents of memory are being updated by more than one process. This eliminates much of the cleanliness of a multiprocess inplementation, since locking must still be used to insure the integrity of the data base.

The results of this implementation, especially the conclusions we draw concerning performance of the algorithm in a high level language, are reported in the earlier part of this report. Details of this project are reported in technical report TR-171 by A. Huber, and in "Further Results with Multi-Process Page Control" by R. Mabee, reprinted in TM-87.

XI. Efficient Processes for the Kernel

As discussed in the previous task description, it appeared that structuring some of the supervisor around separate processes was convenient and appropriate. It was clear, however, that the mechanisms then existing in Multics for the creation and scheduling of processes were somewhat unwieldy for this particular sort of application. We saw many places in the system in which a process could be used if it did not carry with it the full price tag of the user process. In particular we concluded that a process that could take page faults, but could perform no other modifications on its environment, such as adding a new segment to its address space, would be an effective and economical compromise for system processes. We performed an implementation of such a process, in order to demonstrate that its operation was compatible with the Multics structure, and we used this process in a variety of ways. It was

utilized heavily in the design of page control discussed above. It was also used to demonstrate that processes could be used in Multics to handle I/O interrupts. Currently in Multics, the code that responds to an interrupt runs in a very unusual and limited environment, with restrictions such as that it cannot call a locking primitive or perform any other action that might conceivably result in it loosing the processor. If an interrupt could be translated into a wakeup, these problem would vanish. It was clear that the immediate translation of an interrupt into a wakeup was an obvious and crucial idea in the correct structuring of the system. We demonstrated the utility of these fast processes by modifying the teletype interrupt handler so that it ran in such a process. We also explored the use of such a process for handling other I/O interrupts, such as the interrupts necessary to operate our connection to the ARPANET. In the discussion of task XVI below, we demonstrate a structure to the system which provides these efficient processes in a clean and understandable way. The implementation that was part of task XVI ran almost every interrupt handler in the system as a supervisor process.

XII. Multiple Processes in the User Ring

Another related experiment involving the use of multiple processes was the restructuring of the user ring computation so that it could run in a multiprocess environment. While there are a variety of advantages to a multiprocess user environment, such as being able to suspend several commands and then restart them in an order different from the order in which they were suspended, the principal impact on the kernel, as opposed to the user, of multiple processes has to do with handling of the Multics quit signal. The quit signal currently propagates its way through the Multics kernel in a most astonishing and intricate pattern, starting out in an interrupt handler, where

it is translated into a special call to the traffic controller. This call in turn generates a special interrupt in the target process, which may cause that process to run in order to be interrupted. If we understood how to structure the user computations so that the quit was nothing but a wakeup to a separate user process, then the mechanism in the kernel would be much reduced, since the only operation the kernel would perform would be the immediate translation of a quit signal into a wakeup, which is exactly the same action that the kernel would presumably take on any I/O interrupt. A running implementation, of the user computation as a number of processes was produced, although the results of this research were never published. A related document, however, is discussed below in task XVIII.

XIII. Study of Error Recovery

One of the most disruptive events in a system supervisor is the occurrence of an error. An error may be so severe as to cause suspension of all system operation, but even in this context it is necessary to bring the system to an orderly halt so that minimum information is lost. If an error is not that severe, it may still be necessary to reflect the occurrence of this error to some module other than the module that actually discovered the error. It turns out that these error reporting paths are the most intractible communication paths in the system when one attempts to modularize the various functions of the supervisor. Typically, an error is detected at a very low level in the supervisor, and is reported to some higher level, thereby providing a reversed direction communication channel from low to high levels in violation of the layering strategy. During the course of this project we performed a variety of studies with the goal of understanding how Multics should recover from errors, and whether steps taken to insure reliable

recovery from errors might in fact compromise system security. The first project was a study of the Burroughs 7700 operating system, since we were given to believe that this system was highly resilient in the face of errors, and could continue operating without disruption of the user computation. In fact, we concluded after a study of the system listings that the level of recovery provided by the Burroughs system did not markedly exceed that which Multics itself displayed. A more detailed analyses of the various sorts of errors to be expected in the Multics system was performed as part of this project, although the documentation of this report is still in draft form.

A related project which addressed the question of upward communication across layers is described in task XVI.

XIV. Removal of Answering Service from kernel

The Answering Service is that collection of modules that manage the system accounting, authenticate users logging into the system, and keep track of the allocation of typewriter channels and user processes. As currently structured, the Answering Service is a very large collection of code, all of which must be included in the security perimeter of the system. It was our belief that the functions could be structured in such a way that only a small portion required kernel privileges. In fact, we felt that functions traditionally performed as part of the kernel, such as user authentication, could be performed by the user process itself. In order to investigate these beliefs, we developed an alternative structure for the Answering Service that attempted to minimize the kernel functions related to user authentication and accounting. The result of this design was a version of the system with increased flexibility, since users were now permitted to create authenticated and accountible processes at will. At the same time this version reduced the

size of the kernel dramatically, as reported in the earlier portion of this document. A byproduct of this research was increased insight into the relationship between process creation, as currently performed when a user logs in, and the crossing from one protection domain to another, as is often discussed in systems with protection boundaries more general than the Multics ring structure.

A demonstration of this version was implemented. The results are reported in technical report TR-163 by W. Montgomery.

XV. Organization of the Virtual Memory Mechanism of a Computer System

One of the most important results of our research is a method for producing modular, structured software to support the virtual memory mechanism of a computer system. This material is discussed at length in the first part of this report, and is summarized only briefly here.

The method that we propose for organizing a virtual memory mechanism is based on the concept of type extension. A virtual memory mechanism should be regarded as implementing abstract information containers (e.g. segments) out of physical information containers (e.g. main memory blocks and disk records). Further, we showed how one could implement the programs and the address space of the mechanism itself without violating modularity and structure. We illustrated the use of the method by applying it to the redesign of the virtual memory mechanism of Multics.

This work is summarized in the earlier part of this paper and in the Laboratory for Computer Science Annual Report for the period ending June 1976, and is discussed in detail in technical report TR-167 by P. Janson.

XVI. Processor Multiplexing in a Layered Operating System

In the original system, there existed a very intractable entanglement between the virtual memory manager and the processor manager. An important project was to disentangle these two modules, and to produce a structure for the processor manager that was consistent with the principles of layering and type extension developed in the project discussed in the previous section.

The general nature of the entanglement was as follows. The virtual memory manager depended on the processor manager in a number of ways. First, of course, it depended on the processor manager to provide the interpreter for the code of the virtual memory manager. Second, and more explicit, the virtual memory manager called upon the processor manager to suspend the execution of a process that was waiting for a page to be moved from secondary to primary memory. The processor manager, in turn, depended on the virtual memory manager to move to and from main memory the pages containing the description of processes that were about to be run. This unfortunate circularity was eliminated in our redesign by separating the processor manager into two levels. The bottom level was implemented without employing the functions of the virtual memory manager. It executed using only information permanently fixed in primary memory. On top of this layer, the bottom levels of the virtual memory manager ran. The virtual memory manager could call upon this lower level to switch execution from one process to another in order to suspend a process waiting for a page. On top of this bottom layer virtual memory manager, a second layer of processor management was then provided. This upper layer had available to it a virtual memory, and could therefore store the state of a large number of processes, whereas the bottom layer processor manager, since it was restricted to storage permanently allocated in

main memory, could store a state of only a fixed and rather small number of processes. By multiplexing these fixed slots among the larger number of descriptions managed by the top layer processor manager, the effect could be achieved of multiplexing an unbounded number of processes among the available hardware processors.

One additional result of this thesis was a discussion of the problem of upward signalling: the passing of a message from a lower level to a higher level of the system in such a way that the layering dependencies are not violated. The problem arises in this case when, as a result of an event detected by the bottom layer traffic controller, a process whose state is known only at the higher level must be readied for execution. A solution to this problem is proposed which does not make the lower layer processor manager dependent on the uper layer.

This research is discussed in the earlier part of this report, and is presented in detail in technical report TR-164 by D. P. Reed. In order to investigate the performance of the two level processor manager, a detailed design of both levels was completed, and the bottom level was implemented. This detailed design is reported in "A Two-Level Implementation of Processes for Multics" by R.M. Frankston, reprinted in TM-87.

XVII. Separation of Page Control and Segment Control

From the beginning of this project it was clear that one area of great confusion and complexity within the Multics system was the Active Segment Table and the large number of modules that manipulate it. The structure of the Active Segment Table is dictated by the needs of several layers in the memory management system, from page control at the bottom to directory control

at the top. An extensive study was launched of the Active Segment Table and the file system in an attempt to understand what the underlying cause of this entanglement was. A major conclusion of this study was that resource control, in particular the management of storage system quota, was at the root of a great deal of the confusion.

Given the general principles of layering and type extension discussed earlier, it seemed appropriate to attempt to apply them in detail to this area of the system. The particular project undertaken was the separation of the bottom two layers of the virtual memory manager: page control, which moves pages of information to and from main memory, and segment control, which manages the aggregation of pages into segments. These two modules were the primary villans causing the entanglement manifested in the Active Segment Table. The root of the problem was, as expected, resource management, in particular the "quota problem". Much of the structure of the Active Segment Table was being provided so that the low level page manager could implement resource management decisions that reflected policies being specified dynamically by higher level managers. The solution to this problem was to remodularize page control and segment control as three modules rather than two. The bottom layer continued to manage the movement of pages into and out of memory. The top layer provided the abstraction of an active segment, and also the interface to the yet higher layers. The second layer provided an intermediate abstraction that lumped pages together for the purpose of resource control. The result of this particular modularization was a clean isolation of those variables in the Active Segment Table into categories which were referenced by one and only one layer.

This work is reported in technical report TR-177 by A. Mason.

XVIII. Provision of "Breakproof" Environment for User Programming

As various parts of the operating environment are removed from the kernel, the question arises as to where they should be put. If they are placed in the same ring as the executing programs of the user, then they can be destroyed by a programming error of the user. It would be very nice if the removal of programs from the kernel did not lead to a reduced robustness of the programming environment.

This project used the Multics ring mechanism to create an environment which was not a part of the kernel but was still protected from the user. This environment could be used to contain programs private to but still protected from the individual user. We defined a consistent set of programs to constitute this environment, which including the command processor and the error recovery mechanism. The result was a program development and execution environment which was considerably more robust than the current system.

This mechanism was implemented, because we felt we needed operational experience with this subdivision of the user environment into two parts. Much of the Multics environment was easily transformable into this new configuration, although certain components of the system were less tractable than others. The question of how error messages should be signaled in this multi-domain environment was a source of considerable study. There was a slight performance loss in this environment, due to increased page faults from duplication of stacks and related segments in both domains.

This work is reported in technical report TR-175 by H.J. Goldberg.

XIX. Control of Intermodule Dependencies in a Virtual Memory Subsystem

As discussed above in task XV, the techniques of type extension and layering appear to be very important in producing a structured kernel. This

project was a case study of the virtual memory management algorithms of an abstract system resembling Multics, with the intention of applying these principles in such a way that both the number of modules and the number of interconnections between these modules is minimized. The central thesis of this research is that the various operations performed by the layers of the virtual memory manager can be characterized as being of one of two sorts: one that associates and disassociates two computational objects, the other that fetches attributes of a computational object given its name. Decomposition of the virtual memory manager in this way reveals the kind of dependencies that result when one module remembers the name of an object. More strongly, this case study decomposition suggests that if the system provides a primitive mechanism to perform each of these two operations, this pair of operations can be used by several different layers of the virtual memory manager. Such reuse is an especially effective way to reduce the number of modules in a system.

The representation of the operations used in this research is modeled on the LISP concepts of atomic element and property list. The LISP paradigm provides a convenient and suggestive model for the primitive operations performed in this decomposition of a virtual memory manager.

This research is reported in technical report TR-174 by D. Hunt.

XX. New Mechanism for Process Coordination

As part of this project, we proposed a new mechanism for process coordination called "Eventcounts". Basically, Eventcounts are semaphore-like coordination variables that are constrained to take on monotonically increasing values. Coordination of parallel activities is achieved by having a process wait for an Eventcount to attain a given value: one process signals

another by incrementing the value of an Eventcount. Any coordination problem for which a solution has been developed using semaphores can easily be converted to a solution using Eventcounts. In addition, many Eventcount solutions seem to have the property that most Eventcounts are written into by only one process; this reduction in write contention has beneficial effects on security problems and on coordination of processes separated by a transmission delay, as in a "distributed" computer system. Eventcounts provide a solution to the "confined readers" problem, a version of the readers-writers coordination problem in which readers of the information are suppose to be confined in such a way that they cannot communicate information to the writers. Finally, for the class of synchronization problems encountered inside an operating kernel, Eventcounts appear to lead to simple, easy-to-verify solutions.

This work is reported in RFC-102, and in a paper entitled "Synchronization with Eventcounts and Sequencers" to be presented at the 6th Symposium on Operating Systems Principles by D. Reed and R. Kanodia.

XXI. Management of Multiplexed Input/Output

One of the functions of the Multics kernel is to control access to multiplexed I/O streams such as the connection to the front end processor managing terminals or the connection to the ARPANET. The kernel must be involved in the use of these streams, in order to insure that the messages of one user are not inadvertently or maliciously observed or modified by another user. Currently, a large bulk of very complex code is included in the kernel to control each of these streams. This code implements many functions in addition to the necessary kernel function of multiplexing and demultiplexing the messages transmitted over the connection. To reduce the bulk of this

code, we have developed a model of the communication taking place over a multiplexed connection that is general enough to characterize the behavior of the current front end processor, the current ARPANET, and various other protocols for the ARPANET and other nets. From this model it is possible to design modules resident in the kernel that implement the security functions appropriate for any network that can conform to this model, rather than creating a new control program for every network added to the system. A vast majority of the network dependent code can be removed from the kernel and placed instead in the user ring of the individual processes using the network in question.

The model of this portion of the system is rather different in structure than the models proposed to structure the virtual memory manager of the system. The distinctions arise because the I/O stream represents an asynchronous process whose behavior in some sense drives the kernel modules managing the connection. This differing structure may provide an interesting test case for the generality of extended type managers as an organizing tool in a kernel.

XXII. Hardware Estimation of A Process' Primary Memory Requirements

We completed a project to demonstrate that a process' primary memory requirements can be approximated by use of the miss rate on the processor's page table word associative memory. An experimental version of the system demonstrated that the current working set estimator can be eliminated by the use of this hardware feature. The working set estimator is a potentially complex algorithm whose elimination is clearly appropriate in a simplified kernel.

This work is reported in TM-81 by D. Gifford.

PART III: DETAILED STUDY OF POTENTIAL SIZE REDUCTION OF THE MULTICS KERNEL

by

Douglas M. Wells

Abstract

We estimate the impact on the size of the Multics kernel were our various projects carried out. We specify results for three different versions of the kernel. The first includes the effect of those projects that were carried to a trial implementation, or whose size impact could otherwise be accurately predicted. This version corresponds to the estimate stated in the first part of the report. The second version involves projects whose impact could only be estimated. The third version involves a very tentative and unsupported estimate of the impact of producing a file system that only enforces non-discretionary access controls.

Introduction

The first part of this report contained a preliminary study of the impact our project had on the size of the kernel. This section of the report is a detailed analysis of that topic. At the time that the first part of this report was written, the only study of the size of the Multics system was one that was performed at the beginning of the project. For this reason, the numbers reported in the first part of this report are based on modifications to the kernel as it existed at the start of the project. In order to perform a more detailed analysis, we examined the kernel as it exists in the standard system now. Since the size of the standard system has increased since the

start of our project, the absolute numbers reported in this portion of the report differ from those given in the preliminary study. The percentage impacts that we report are approximately the same, however.

Scope of the Kernel

In defining the security kernel considered for this work, we consider several parts of the standard system:

- Ring 0 supervisor. Potentially, any procedure executing in ring 0 can examine or modify any part of Multics. We therefore need to consider any module included in ring 0 as part of the security kernel.

- Message Segment Primitives. For purposes of the Access Isolation Mechanism, message segments may contain information at multiple levels and/or categories. Thus, any misbehavior on the part of the message segment primitives could allow unauthorized access to data.

- Answering Service. Because the Answering Service is responsible for the creation of all other processes, an error here could cause a process to be created with uncontrolled privileges.

- Backup Services. One of the fundamental services of Multics is providing reliable file storage services. Any error in one of these services could cause a segment to be reloaded at a level other than its proper level.

- Detachable Storage System Media. Although Janson's type extension techniques indicate methods of handling these outside the kernel, the actual Multics implementation is new enough that there has not been an actual analysis of it. We will therefore consider the existing mechanisms as being within the security kernel.

There are also several areas that are not considered here:

- I/O Services. Although bulk printer and card services are a service of the standard Multics, we have not devoted resources to studying this area. Primarily, this is because the problems in the area seem not to be ones that require engineering of the kernel software, but rather ones of adhering to various laws and government regulations concerning classified data. In addition, we believe that these services might better be performed in a Secure Front End Processor, as described in scenario two. We will, therefore, consider I/O services as being outside the security kernel.

- Frontend Network Processor. The standard Frontend Network Processor (FNP) does not seem well suited to interfacing a Multics security kernel. The problem areas include a lack of protection hardware and a poorly structured hardware interface between the FNP and the Multics memory. Because of these problems, we have not pursued the use of the FNP in a secure version of Multics. Rather, we have assumed that some Secure Front End Processor with its own security kernel is used for system Input/Output. The use of this SFEP is discussed further in scenario two below.

- Special Backup Services. The standard Multics system provides two backup services that will not be considered in this report: complete dumps of the hierarchy, and retrieval of individual segments from backup tapes. We believe that these services need not be considered here because they are each only an optimization of one of the other services. A complete dump is taken only to coalesce the results of all previous incremental

dumps. A retrieval is essentially a reload in which most of the segments that would normally be reloaded are skipped.

Base Multics Kernel

In assessing the impact of this project upon the size of the Multics kernel, we report the results of the various simplifications upon a base system. The base system we have chosen is the Multics system actually running at the time that this report was written. This system, designated system MSS 31-6, was installed at M.I.T. on June 23, 1977. This system is typical of the various versions of Multics that have existed during the period of this research. One slight peculiarity of this system is that it contains two versions of the Backup mechanism. The older, now-obsolete Backup system (the Hierarchy Dumper and Reloader) will be retained until confidence in the newer Backup system (the Volume Dumper and Reloader) is acquired.

At the outset of this research, an initial census was made of the Multics ring 0 supervisor [Voydock, in Clark, 1977*]. That census included a functional breakdown of the modules in the ring 0 portion of Multics, and provided totals of the sizes of the programs. Then, based on an assumption of 5 words of text section per PL/I source statement, the size of the ring 0 part of the system was estimated at 44,000 source lines. That number, plus the 13,000 lines in the Answering Service is the basis of the 57,000 line kernel size used in part one of this report.

We also performed a census upon our base system for the purpose of determining the sizes of the various functional categories. For this census,

* References in this part of the report may be located in the Publications and References sections of part 1.

we counted the sizes of the modules, both the text section size and the count of source statements. The differences between that 1974 version of Multics (MSS 20-10a) and our base system (MSS 31-6) reveal a few interesting changes. The ring 0 portion of the system has increased in size by 48%, from 157,000 words of text to 233,000. The number of source modules has increased in almost the same proportion, 305 versus 432. Only one major section of the system has crossed the ring 0 boundary: Tape Control has been moved outside the supervisor.

On the other hand, a surprising number of things have remained the same. It appears that the sizes of the individual modules have remained relatively constant, averaging about 525 words of text section per module. Due to the differences in methods of computing program size, we can't directly compare the relative usage of assembly language, but we do find that the proportion of assembly language modules is about the same in the previous system as in the base system. Also, it should be noted that there are no major new functional units in ring 0; the only changes have been ones of replacement or alteration.

If we look for the reason for the increase in the size of the ring 0 portion of the system, we immediately find that the capabilities of the system have been improved substantially. During the period since the initial census of the Multics supervisor was performed, the system has been altered in a number of significant ways:

- the Access Isolation Mechanism has been incorporated into the system,
- a "new" storage system implementation has been installed, including support for detachable parts of the hierarchy,

- the salvager, which was previously a stand-alone system, is now an integral part of the normal Multics,

- PL/I support of language I/O features has been dramatically improved including a reimplementation of the PL/I "file" support,

- dynamic reconfiguration has been "idiot-proofed",

- a rewritten typewriter-control system has been installed.

Typically, each of these reimplementations has caused the size of the subsystem to increase. There are two primary reasons for the increase -- expanded function, and improved debugging and metering facilities within the subsystem.

The base system is organized so as to simplify system maintenance and development, not to reduce the size of the kernel portion of the system. One result of this organization is that the kernel service processes, such as the Answering Service, tend to use normal system utility routines. These utility procedures often include more function than is needed by the service process. An example of this is the temporary segment manager. Although a temporary segment can be created with only one PL/I source language call, the temporary segment manager maintains a pool of such temporary segments in order to eliminate unnecessary costs of segment creation and deletion. Because the use of this facility can improve overall system performance, the Answering Service uses this (and other) facilities.

An unfortunate result of this organization is the fact that the address space of the non-ring 0 kernel processes is much larger than it needs to be. Many system utilities, even those not used by the kernel processes, are

included in segments in their address spaces. In certifying the standard system, however, all these extra modules would have to be audited. In performing the census of the base system, we chose not to include all these extra modules. First, the actual identification and analysis of these modules seemed impractical. Second, the system could be trivially recoded to eliminate such uses. Therefore, in computing the size of the kernel of the base system, we have applied one exclusion factor.

The rule we have applied has been: if the call to the utility procedure could be replaced by fewer than about 10 lines of code in the original program, we have not counted that utility as being in the kernel. We estimate that had we included all those extra modules in the system, the base kernel would have been about 20,000 lines larger.

To give some indication of the sizes of the various subsystems of the base Multics system, we will give a functional breakdown of the components. The numbers given here are for all source modules in the kernel that contain executable code. That is, modules that contain only functional parameters or table space are excluded from the count. Gate segments have also been excluded. The SIZE is a count of the source language statements in the procedure modules. NON-PL/I is the percentage of the code, measured in source statements, written in a language other than PL/I. It should be noted that dispatch modules are typically coded in assembly language or macro language and artifically increase this percentage when used as a measure of the use of non-higher-level languages. The TEXT-LEN section indicates the size of the "text" section of the object modules. For PL/I, this includes all constants and executable instructions. Although some assembly language programs may

contain executable instructions in other sections of the object module, this number provides a good indication of the size of the object program.

CATEGORY	MODULES	SIZE	NON-PL/I	TEXT-LEN
Initialization	45	4636	37%	23708
Reconfiguration	13	1126	2%	7714
Fault Handling	13	1326	90%	2158
I/O Control	38	3526	28%	17598
Printer	6	958	73%	2532
Tape Control	21	2662	2%	10449
TTY Control	19	4266	5%	20477
ARPANET	55	7493	1%	40338
Error Handling	25	2016	9%	9312
Process Control	28	1296	6%	8773
Traffic Control	3	2296	92%	2710
IPC	25	3061	2%	16160
Process Signals	5	390	17%	1450
Resource Control	32	2343	---	11342
Storage System	38	5366	1%	35864
Directory Control	51	6609	< 1%	34434
Segment Control	32	1973	5%	11460
Page Control	26	5870	69%	13704
Salvager	18	2897	1%	20747
Dynamic Linker	14	1793	11%	7234
File System	5	1161	2%	6631
AIM	7	924	6%	6223
Error Interpretation	12	856	1%	7228

Kernel Utility	5	470	93%	663
Shared Utility	16	2800	81%	5069
Backup	44	7827	---	57002
Answering Service	73	12987	2%	94609
PL/I Support	39	12504	94%	18641
Miscellaneous	4	191	22%	918
Totals	712	101623	26%	495148
(Ring 0 Only)	432	61848	41%	232824

Since the total number of source lines involved is about 100,000, each thousand lines represents about 1% of the kernel size. It is useful to keep this comparison in mind while reading the following description of size reductions. As the reductions accumulate, it is also useful to remember that the perceived impact measured in terms of the final kernel is much larger. Thus, a removal of 1000 lines would reduce the final 38,000 line kernel by 2 1/2%, not 1%.

First Level Reduction Estimates

This scenario includes those concepts whose feasibility has been proven and that have little or no impact upon the user interfaces to the system. The changes described at this level would reduce the size of the kernel by 40% and could probably be done in one year real time.

The changes in this version of the kernel include:

- removal of obsolete code,

- removal of extraneous PL/I support routines,

- restructuring of page control,

- removal of Answering Service,

- use of encipherment in backup services,

- making ring-0 and the kernel coincident,

- removal of dynamic linker and reference name management,

- miscellaneous cleanups and recoding in PL/I.

Removal of Obsolete Code

As the Multics system has evolved over the years since its inception, many subsystems have been redesigned and now have significantly different interfaces. Often, the newer interfaces are more primitive (and therefore simpler) than the old interfaces. In order to provide compatibility to the existing user community, the old interfaces are usually recoded to use the newer interfaces. These write-arounds are then made a part of the newly redesigned subsystem. For subsystems that are a part of the kernel, the write-arounds are also included in the kernel.

Early experiments with removing the dynamic linker from the kernel have indicated that moving the write-arounds outside of the kernel can usually be done quite trivially by replacing the kernel gate procedures with non-kernel dispatch modules. Users would call the entry points in these modules, which would then transfer to the actual kernel gate procedures or to the write-around as appropriate to the particular function invoked. This approach

is not usually followed since old application programs that make use of the write-arounds will encounter a slight performance loss in going through an extra level of name resolution and by the addition of one extra page to the working set.

Portions of the base system that include significant amounts of obsolete code include: Backup with 3400 lines, Tape Control with 1000 lines, PL/I Support with 3000 lines, and Directory Control with 700 lines. In addition, there are small amounts in various other subsystems that total about 1500 lines.

Net reduction: 8100 lines

Removal of Extraneous PL/I Support Routines

In order to reduce the size of object programs and in an attempt to provide a higher degree of compatibility, the PL/I compiler makes heavy use of run-time operators. This is especially true for Input/Output support and mathematical functions. Currently, all of these operators are combined into one large segment that is included in the kernel. Fully 55% of this operator segment in the base system is never needed by kernel procedures. Also, most of this support code in written in assembly language. Thus, the removal of these routines would have a significant impact on the size of the kernel.

25 modules involving 5200 lines of non-obsolete source could be directly removed from the kernel. In addition, approximately 700 more lines could be eliminated from kernel support modules and moved to new, non-kernel modules.

Net reduction: 5900 lines

Restructuring Page Control

Page Control is one of the most complex subsystems in the base Multics system: A given page may be in any one of about thirty states. Most of the state transitions occur during the handling of faults or interrupts. More than two-thirds of it is coded in assembly language. Thus, it seemed an ideal candidate to test our ideas about use of kernel processes and conversion of assembly language programs to PL/I.

Due to our special interest in the effects of using multiple processes and recoding in PL/I, we made an attempt to optimize and tune this multi-process version of Page Control. As reported in [Mabee, in Clark, 1977], the final version consumes 50% more CPU resources in managing pages than the equivalent assembly language version. The object modules are also about 20% larger. On the other hand, when measured in source statements, the PL/I version is 1000 lines smaller, about 17% of the size of Page Control.

An analysis of the functioning of this subsystem indicates that the poorer performance is almost entirely due to the recoding in PL/I, not to the use of multiple processes. Thus, even in cases where the performance of the system is critical, the use of multiple processes to allow a simplified structure does not seem to intolerably degrade the performance.

In addition, there are a number of functions in page control that could be removed without seriously decreasing the performance of the system. Although often the amounts of code that would be removed are not large, these functions unnecessarily complicate the transitions within page control. These functions include: aborting read/write sequences while moving pages from the bulk store to the disk, special-casing segment truncations.

One significant point about page control is that it is one of two portions of the kernel that are coded in assembly language primarily for efficiency. Although the implementation of multi-process page control was converted to PL/I and thus was less efficient, the overhead attributable to page control (whether coded in PL/I or assembly language) can be reduced to an arbitrarily low amount by using large memory hardware configurations.

The implementation performed by Huber [Huber, 1976] demonstrated that 1000 source lines could be removed from Page Control.

Net reduction: 1000 lines

Removal of Answering Service

The Answering Service is one of the largest single components of the base kernel. By itself, it comprises 13% of the kernel. In addition, significant portions of ARPANET and TTY Control are included in the kernel only because they are required by the Answering Service (or Backup, the other service process included in the kernel). Thus, reductions in the size of the Answering Service have enhanced effects on the kernel.

The trial implementation by Montgomery [Montgomery, 1976] demonstrated that the Answering Service could be divided into two parts: modules that managed the creation and access capabilities of processes, and modules that interact with users in order to call upon the process controlling modules. Those modules that only interact with users can be moved outside the kernel.

Based upon the results of that implementation, we find that those modules that manage user processes comprise less than 7% of the size of the Answering Service -- resulting in the elimination of over 12,000 lines of code. This

alone reduces the size of the kernel by 12%. In addition, 2700 lines of ARPANET support code included in the base kernel are used only to interact with users. Because the corresponding portions of the Answering Service have been removed from the kernel, this ARPANET code can also be removed.

Net reduction: 14,700 lines

Encryption in Backup Services

As part of the implementation of the "new" storage system, the Multics backup mechanism has been changed to use a different mechanism for determining which segments require backing up. The previous mechanism used a privileged process that periodically scanned the storage system hierarchy looking for segments that had been modified since the previous such scan of the hierarchy. In the new mechanism, the storage system notices whenever a segment has been modified and notifies the Backup process. The Backup process then copies that segment onto tape.

Using a methodology similar to that applied to the Answering Service, we can divide the new backup mechanism into two parts: those modules that interface to the storage system, and those that perform external functions such as actually writing the information onto tape, or producing human-readable maps of the backed-up data. By enciphering the segments and associated storage system information as it is passed out of the kernel, we can remove the external functions from the kernel, leaving only a small storage system interface still in the kernel. In return, we would have to add the enciphering mechanism.

It should be noted that the use of encipherment here is the first instance we have proposed for actually allowing a non-kernel module to physically maintain a copy of a particular protected object. The only mechanism used here to ensure security is the fact that the data is enciphered. We are relying on the extreme difficulty of decoding the data. Although there are no known proofs of "uncrackability" of existing "difficult" encipherment schemes, there are claimed to be encipherment algorithms that have been certified as acceptable for use at any desired level of security.*

The elimination of the external functions allow us to reduce the size of the kernel portion of Backup to about 1100 lines, a reduction from the original 2800 lines of non-obsolete code. Implementations of enciphering mechanisms for other purposes have indicated that we would have to add about 500 lines to the kernel to perform the encipherment and to manage the cipher keys (or to manage flow of data to and from an enciphering box). Also, in order to allow the system security officer to inspect the backup tapes and request non-standard retrievals, we would probably need another 500 lines of code.

In addition, because the Tape Control programs no longer need to be considered part of the kernel, we can eliminate another 1600 lines of tape management code from the kernel.

Net reduction: 2300 lines

* Kahn, D., The Codebreakers, Macmillan, New York, 1967.

Making Ring-0 and the Kernel Coincident

The Multics ring 0 is a very special environment within Multics. Many aspects of the environment while running in ring 0 are special cased: all programs in ring 0 are pre-linked at system initialization, so the dynamic linker is not required; the segment number of any given segment is the same in all processes, so linkage sections can be shared; ring 0 segments are never deactivated, so segments faults do not happen on kernel segments. All of this means that programs executing in ring 0 exist in a more primitive environment than programs executing in other rings. In fact, there are a number of kernel subsystems that only will work for outer ring callers. For the subsystems that are used by both ring 0 and outer ring programs, however, we find that there are often two versions of a particular function, one for ring 0 and one for the other rings. For example, there is a prelinker program for ring 0 and a dynamic linker program for the other rings; there is a program that initially activates ring 0 programs, and another program that activates segments in response to segment faults by outer ring procedures.

When trying to reduce the size and complexity of a security kernel, we find that the duplication of functions unnecessarily increases the size of the kernel. If we can remove the outer-ring version of a program from the kernel, we often eliminate more than half the statements in the overall subsystem. In order to eliminate the outer-ring version of the program from the security kernel, however, we must move all kernel programs into ring 0. Thus, to allow the removal of these duplicate functions, we need to move the message segment primitives, the detachable media manager, and the appropriate parts of the Backup and Answering Services processes into ring 0.

In addition to the linker and reference name table management code described below, we can eliminate 800 lines of I/O System, all the File System, all the Error Interpretation system, 260 lines of Process Signal code, and all of TTY Control that is in the outer ring.

Net reduction: 3240 lines

Removal of Linker and Rererence Name Table Management

Early implementations by Janson [Janson, 1974] and Bratt [Bratt, 1975]] demonstrated that the dynamic linker and reference name table (RNT) management were functions that could easily be removed from the ring 0 portion of Multics. Unfortunately, these subsystems were still required by privileged processes such as the Answering Service and the Backup processes, and as such, had to be included within the security kernel of the system. With the changes to the Answering Service and Backup functions described above, however, the remaining kernel functions could easily be moved into ring 0 using kernel processes as described in the discussion of Page Control above or the equivalent hardcore processes available in the base version of Multics.

The removal of the dynamic linker and RNT management from the kernel allows us to remove 1950 lines of code. Furthermore, these particular functions include a disproportionate number of entry points into the kernel. Thus, removing these two functions also significantly reduces the complexity of the interface into the kernel.

Net reduction: 1950 lines

Miscellaneous Cleanups

There are a number of other removals that will be listed here. Typically, these are straight forward cleanups that have not been performed on the standard Multics due to the necessity of changing large amounts of other kernel programs to replace calls to the eliminated functions. The code conversion and canonicalization portions of TTY Control can be easily moved outside the kernel. The full implementation of IPC channels is not needed in the kernel; the "special" channels will handle all needs for IPC by kernel functions. There are a number of Directory Control functions, such as make_seg and move_seg, that need not be in the kernel. The Fault Handling modules translate hardware faults into the equivalent PL/I faults even though not kernel functions depend on this translation. There are also a number of places where modules can be converted from assembly language to PL/I without significantly affecting the performance of the system.

The implementation of these cleanups should result in the removal of about 3000 lines of code.

Net reduction: 3000 lines

Summary of Level One Reductions

After performing this first set of simplifications, we have a system that provides essentially the same user interface as the base system. Only in rare circumstances would even the side effects of the functioning be different. The only essential difference would be the fact that the kernel would be some 40% smaller than the base system. The breakdown by category, including the change from the base system, is as follows:

CATEGORY	SIZE	% CHANGE
Initialization	4636	--
Reconfiguration	1126	--
Fault Handling	1326	--
I/O Control	2783	-21%
Printer	958	--
Tape Control	0	-100%
TTY Control	3100	-27%
ARPANET	4824	-36%
Error Handling	2016	--
Process Control	1200	-7%
Traffic Control	2200	-4%
IPC	2400	-21%
Process Signals	120	-69%
Resource Control	2343	--
Storage System	5366	--
Directory Control	5900	-11%
Segment Control	1973	--
Page Control	4900	-17%
Salvager	2897	--
Dynamic Linker	100	-94%
File System	0	-100%
AIM	924	--
Error Interpretation	0	-100%
Kernel Utility	470	--
Shared Utility	2800	--

Backup	1600	-80%
Answering Service	1000	-92%
PL/I Support	3600	-71%
Encryption	500	New
Miscellaneous	113	-41%
Totals	61075	-40%
(Base System)	101623	
(Reduction)	40548	

Level Two Reductions

This level of kernel revision includes those concepts that would either alter the function of the system in some manner that would show up at the user interface, or concepts that require significantly more work than those in the first scenario. Because this scenario includes a number of concepts for which we have not completed trial implementations, the estimated size of the resulting kernel is much less precise.

The changes incorporated in this version include:

- two level traffic controller.
- revised initialization
- simplification of Directory Control interfaces.
- separation of tracing/metering code.

- use of kernel processes for multiplexed I/O

- use of Front End I/O processor.

Two-Level Traffic Controller

By using the same methodology described above for the Answering Service, Reed [Reed, 1976] was able to divide traffic control into two parts -- one implementing basic mechansims, the other higher level policies. In this case, also, we were able to move the policy manager outside the security kernel. By restricting the outer-ring mechanism to the control of scheduling parameters, we can ensure that it cannot cause the leakage of protected information. In fact, the particular mechanism proposed allows us to move almost all the base system's process controlling subsystems outside the kernel.

The trial implementation of the lower level virtual processor manager took 1176 source lines. This implementation did not include the functions necessary to allow the higher level process manager to cause switching of user processes, but it did indicate that the addition of that function would only add about 600 lines to the kernel modules. This small amount of kernel code, together with the proposed (non-kernel) "level 2" policy mechanism would completely replace the base kernel functions of Process Control, Traffic Control and Process Signalling. In addition, it would eliminate 952 lines of Fault Handling, and all of IPC except the message segment primitives.

Net reduction: 3500 lines.

Core Image Initialization

The base version of Multics initializes itself by having a small bootstrapping program loaded into primary memory. This small program then incrementally reads more of the Multics system from a tape. Some of this newly-read system serves only to provide an interim environment for loading the actual programs that will actually function in the fully-operational Multics environment. If we could just load a completely initialized image of the Multics system, we would eliminate a number of these initialization programs from the kernel system.

In examining the problems associated with this type of "core image initialization," Luniewski [Luniewski, 1977] found that the major problem area was one of adjusting the size of various databases. A trial implementation showed that these tables could be dynamically grown at the expense of adding about 500 lines of reconfiguration code to the system. In return for this, we can eliminate 2500 lines of initialization code; much of it in assembly language.

Net reduction: 2000 lines

Simplification of Directory Control Interfaces

In examining some parts of the system, we find large portions of the subsystem are used to provide interfaces to the user. Typical systems in which this is true include Input/Output Control and Directory Control. This is especially true of Directory Control, because of the large number of attributes that are handled: Access Control Lists, Time Last Modified, Safety Switch, Copy Switch, etc. The base version of Multics has a separate entry into the kernel for reading each of these values, and if the particular value

is settable by the user, a separate entry for setting the value. At each of these entries, the kernel program must first verify the arguments, then verify that the particular operation is allowed for this user, and finally retrieve or store the appropriate value in the directory. We find that much of the code in these operations is used in the verification of the arguments and the access.

If we were to reduce the number of entries so that there was essentially one entry for each type of access that was allowed, we could save much of this duplicated code. In the case of Directory Control, we would have one entry to read the current Access Control List, another to replace the entire list. We would have one entry to set the various switches and parameters; there would be another entry that would return the value of the switches and the various times stored by the system.

In the case of directory listing, we find that the base interface uses a "star name" as an argument and tests each name in the directory against the star name to see if it matches. Also, there are various entry points to return additional information (such as the times and effective access) for each returned entry. A much simpler and smaller interface would return all entry names in a directory and require that the star name processing be done outside the kernel. If the extra information were desired, the kernel interface should be designed so that it always returns the extra information that was most often used. Other, atypical cases could use the status returning entry described above to get any additional desired information.

One other simplification possible in this area is the elimination of the use of the PL/I area functions for returning this information. If we change

the information returning entry points, so that they write into a preallocated buffer, rather than allocating in an outer-ring area, we can eliminate the area management code from the kernel. This would remove about 800 more lines from the kernel.

When this simplification scheme is applied to the base Multics system, we find that over 1000 lines could be saved in this user interface area. Even more importantly, the user interface area is one that often contains security leaks because of errors in programs that incorrectly validate arguments*. Thus, by reducing the size and complexity of this particular area, we have made extra progress in aiding the auditing process.

On the other hand, because the kernel no longer performs complex interface operations, certain actions that are possible under the base version of Multics can no longer be performed. The actions that would be disallowed correspond to the case where a user has only "append" access to a directory. Thus, simplifying the interfaces as described here essentially requires that we remove the concept of "append only" directories. Since the concept is only occasionally used, and is often replaceable by use of "add only" message segments, the loss does not seem to affect the normal capabilities of Multics.

Net reduction: 1800 lines

An investigation into known security leaks in earlier versions of Multics [Janson and Forsdick, in Clark, 1977] showed that most leaks in the system could be categorized into a very small number of areas. One such area was the improper validation of arguments.

Separation of Debugging/Metering Code

As mentioned in the introduction to this section, one of the areas that has grown most in the time since our initial census of the system, is the area of debugging and metering. This is primarily due to the intense effort being made to further develop Multics and to improve its performance. Many kernel subsystems now include extensive tracing and performance measurement facilities. Unfortunately, these facilities are undesirable in a kernel that is to be audited. By definition, the code performs no part in effecting the desired functions. Contrarily, the only non-transparent actions possible are deleterious. On the other hand, if there are ever to be future improvements to kernel system, this debugging code would prove to be very useful. Thus, we would propose a compile-time feature or a load-time feature that would allow the debugging and metering code to remain in the source code, but would guarantee that the code could not affect the security kernel. One example of how this could be done is to consider adding a new section to the object module. Although normally present for debugging runs, etc., the security kernel version of the system could discard this section of code, replacing it with no-operation instructions.

The addition of such a feature would allow the elimination of about 1500 lines from the system. Because of the removals allowed by the Two Level Traffic Controller, however, only about 300 lines of this represents debugging and metering code that would otherwise still be in the kernel. Since the debugging and metering code is widely distributed through the system, its removal would tend to reduce the size of many modules rather than eliminate a few of them.

Net reduction: 300 lines

Use of Kernel Processes for Multiplexed I/O

The control of multiplexed devices, such as an ARPANET or a typewriter controller, is conceptually a simple task. In one direction, data is accepted from a user, inserted into a queue, and then, when the device is ready, the data is transmitted to the I/O device. In the other direction, data is accepted from the device, and then placed on a queue for a particular user as indicated by a field in a message header. Yet we find that the components for handling multiplexed devices make up almost 20% of the base system. Obviously, something is more complicated than it appears at first glance.

When we analyze the existing software, we find that none of the multiplexed I/O subsystems adhere to this simple model. In fact, the size of the particular subsystem seems to be in direct proportion to its deviation from this model. The problem seems to be in controlling the flow of data to each individual device. In the case of typewriters, some are much faster then others. So, if fast devices had to wait for slower devices to complete processing, the faster devices would spend most of the time waiting.

The implemented solution to this problem in the TTY Control and ARPANET subsystems is to also keep an output queue for each user. Data from these individual queues is entered into the actual device queue only when the user's subchannel has indicated that it will accept the data. Unfortunately, the signal that the subchannel will accept the data is received asynchronously and there may be no user process available to process the input. Due to historical reasons of efficiency and the difficulty of creating processes, the solution generally employed to solve this problem has been to process the input data during the handling of the device interrupt signal.

Unfortunately, performing this processing at interrupt time unduly complicates the algorithm. There are several reasons for this. First, the normal locking primitives can not be used while processing an interrupt; the process which has a lock set may be the same one that is processing the interrupt. Second, there must exist code to do the same function in a normal user process; if the device is quiescent, there will be no interrupts coming in, so a call-side process must initiate the operation. Third, all programs and data that are referenced at interrupt time, must be in wired-down locations in primary memory; thus, often there are two types of queues -- one wired and the other pageable.

For all these reasons and more, architectures that use a dedicated device process are usually simpler than those that use interrupt-time processing. Furthermore, experiments by Ciccarelli [Ciccarelli, 1977] indicate that when a process structure is used, the various multiplexed I/O device subsystems can share buffer management primitives. Since the buffer management is one of the largest components of each of the base I/O subsystems, the use of common buffer management would greatly reduce the bulk of the system. Although we have no firm figures, initial estimates are that the use of kernel processes and the associated sharing of buffer manager primitives would probably eliminate 6000 lines of source.

Net reduction: 6000 lines

Use of Front End Processor for I/O

Approximately 19% of the base Multics kernel is devoted to controlling source/sink Input/Output devices, such as teletypewriters, printers, and tape drives. On the other hand, the view from outside the kernel is that these

various peripheral devices are essentially interchangable. This is evidenced by the fact that normal usage of these devices is via an I/O switch, which provides essentially three types of operation -- read, write, and special-function. If we could move the actual device control to another, dedicated-purpose processor, we could eliminate a significant portion of the kernel. In order to communicate with this SFEP, the Multics kernel would retain only one program, a multiplexed I/O handler as described in the previous section.

At first glance it appears that we are only moving the functionality from one security kernel to another in the Front End Processor. There are, however, several advantages to moving the functionality to the SFEP. First, the SFEP is a dedicated machine. There are no users writing programs to try to "crack" the system. Second, the SFEP gets its commands and data via "thin-wire communications". Because the user and the SFEP are using different address spaces, the commands and data are delivered from the caller to the SFEP as complete, integral messages. Because the message delivered to the SFEP cannot be changed by the caller, the SFEP does not need to consider the problems that occur if a user is allowed to change the data after it has been validated. Third, and most important, there is no need for sharing of data or for communication between the individual device drivers. Because of this, the system can be completely compartmentalized. Other than a small kernel devoted entirely to message switching and primitive I/O operation validation, the various device control programs can be entirely separated from one another. Thus, except for multiplexed devices, the device control programs do not have to be certified and are not part of the security kernel(s) of the complete system.

Although we have not investigated the actual size of the kernel for the SFEP, implementations by other researchers have indicated that it should not be more than 1000 or 2000 source lines [Lipner, 1974]. The savings in the Multics security kernel, on the other hand, would be on the order of 3000 lines.

Net reduction: 3000 lines

Summary of Second Level Reductions

After performing the above modifications, we have a Multics kernel that is only 44,500 lines, some 44% of the size of base kernel. The only major change at the user interface is the lack of "append only" access to directories. An approximate breakdown of the subsystem sizes is shown below:

CATEGORY	SIZE	% CHANGE
Initialization	2100	-55%
Reconfiguration	1600	+42%
Fault Handling	374	-72%
I/O Control	2300	-35%
Buffer Management	1000	New
Printer	0	-100%
Tape Control	0	-100%
TTY Control	0	-100%
ARPANET	0	-100%
SFEP Control	1000	New
Error Handling	2016	--
Process Control	0	-100%

Traffic Control	600	-74%
IPC	1563	-49%
Process Signals	0	-100%
Resource Control	2343	--
Storage System	5366	--
Directory Control	4700	-29%
Segment Control	1973	--
Page Control	4800	-18%
Salvager	2897	--
Dynamic Linker	100	-94%
File System	0	-100%
AIM	924	--
Error Interpretation	0	-100%
Kernel Utility	470	--
Shared Utility	1700	-29%
Backup	1600	-80%
Answering Service	1000	-92%
PL/I Support	3400	-73%
Encryption	500	New
Miscellaneous	113	-41%
	------	-----
Totals	44439	-56%
(Base System)	101623	
(Reduction)	57184	

Third Level Reductions

The final scenario presented here provides the most significant simplification of the kernel. Correspondingly, it also requires more drastic changes to the structuring of the system and presents a user interface that has significantly different side-effects than the base kernel.

The changes proposed at this level include:

- removal of Discretionary Access Controls.
- restructuring of the Salvager.
- separation of Segment Control and Page Control.

REMOVAL OF DISCRETIONARY ACCESS CONTROLS

The underlying security model does not require that discretionary access controls be included in the security kernel. In fact, only the non-discretionary access controls, the segregation into levels and categories, needs to be enforced to ensure that security is not compromised. In Multics, the discretionary access controls are the Access Control Lists, which are managed by Directory Control. If Directory Control can be moved outside the kernel, the kernel will have shrunk by a substantial amount.

In the base version of Multics, however, Directory Control also manages the non-discretionary access controls. The particular access authorization of any particular segment or directory is stored in the parent directory. To move the discretionary access controls outside the kernel would require separating the base Directory Control into two parts: one to manage the discretionary access controls and one to manage the non-direcretionary

controls. This leaves us with only a minimal, kernel directory system which manages only the contents of segments and the access authorizations of those segments inside the kernel. Everything else that was previously in Directory Control has been moved outside the kernel, though probably running in a more privileged ring than normal users.

One method of implementing this minimal kernel directory system calls for the use of a linear directory (somtimes called a "flat file system") inside the kernel. In fact, this system would appear to be much like the "inode" list of the UNIX system [Ritchie and Thompson, 1974]. The kernel would provide facilities to create, segments, to hand over use of the pages in the segment to Page Control, to delete the segments and to upgrade the segments. The non-kernel Directory Control would use some of these segments as directory catalogs, and store knowledge of other segments in these directory catalogs. Rather than containing disk addresses, the directories would contain unique ids as generated by the kernel directory system.

The removal of this particular part of the base kernel, however, would have several adverse effects. First of all, the user interface to the Directory Control system would change substantially. There are currently a few aspects of the AIM system that require that processes running at multiple levels be able to read and write in a particular segment. The current use of multi-level message segments is an example of this. Either there would have to be invented a special mechanism for full-duplex communication between processes at multiple levels, or each particular use would have to be special cased inside the security kernel. In either case, the user interface to the mechanism would be substantially changed.

A second adverse effect is that the Directory Control system, including normal Multics Access Control Lists, would no longer be certified. Thus, although the new, smaller kernel would adhere to the underlying security model, there would still be the possibility that one user could obtain unauthorized access to another user's data due to a bug in the non-kernel Directory Control system. For this to happen, the two users would have to share a category, but there are currently fewer than 72 categories. Thus, for any system with more than 72 registered users, at least two of those users would have to share a category. Any protection features between the two users would be by the possibly-uncertified discretionary control system, not by the kernel.

We have not performed any trial implementations of this concept, but initial analyses indicate that about 85% of the remaining Directory Control could be moved outside of the security kernel.

Net reduction: 4000 lines

Restructuring of Salvager

If the Directory Control system were moved outside the kernel, the Directory Salvager would have to be divided into the same two functional components. The kernel component would have to be able to reconstruct the kernel segment list. It would ignore the contents of the segment even if the non-kernel Directory Control was using that particular segment as a directory catalog. The other salvager would perform most of the functions that the base system salvager -- reconstructing ACLS, validating entry name chains, rebuilding hash tables, etc.

Because the kernel version of the Salvager would have such a simple job, initial indications are that it would require about 15% of the code of the present Salvager, resulting in a savings of about 2400 lines of code.

Net reduction: 2400 lines

Summary of Level Three Reductions

Because of the lack of firm numbers for this last scenario, we will simply suggest that this final system results in a security kernel which is approximately 38,000 lines, some 37% of the size of the base kernel. The following table is indicative of the sizes of the components: some 37% of the size of the base kernel.

CATEGORY	SIZE
Initialization	2100
Reconfiguration	1600
Fault Handling	400
I/O Control	2300
Buffer Management	1000
SFEP Control	1000
Error Handling	2000
Traffic Control	600
IPC	1400
Resource Control	2300
Storage System	5400
Directory Control	700
Segment Control	2000

Page Control	4800
Salvager	500
Dynamic Linker	100
AIM	900
Kernel Utility	500
Shared Utility	1700
Backup	1600
Answering Service	1000
PL/I Support	3400
Encryption	500
Miscellaneous	200

Further Reductions

Although we have now reduced the kernel by 63% of its base size, there are, in fact, a few subsystems that we have not analyzed here. One is the Storage System. It would appear that something could be done to allow the separation of the detachable disk management code to be divided in much the same way that the Answering Service and the Traffic Control sections were.

Another major area not analyzed here is the separation of segment control and page control. This is a project that we propose primarily because it would improve the modularity of the two subsystems, and thus decrease the complexity of the system. Currently, Segment Control and Page Control share a data base -- the Active Segment Table. The fact that these two subsystems share this database disproportionately increases the difficulty of verifying the two systems. Rather than considering two reasonably small subsystems that communicate via normal subroutine call interfaces, an auditor of the base

system (or even of any of the other scenarios above) would have to consider the system which consists of the union of the two subsystems. If we assume that difficulty of auditing is proportional to the possible connectivity within a system, we find that the difficulty increases as about the square of the size of the system. Using this assumption, we find that the auditor would be faced with a task twice as difficult as necessary.

We have no indication of the size of the resulting code. It is quite possible that the two resulting subsystems would be substantially larger than the existing systems. But we tend to doubt that this would happen. Trial implementations of other subsystems have shown that although the first rewrite is larger than the initial system, a few passes through the algorithms often realize substantial improvements, both in the size and the speed of the code.

The Resource Control Package is another major section that we have not considered. Because this subsystem both controls the access of peripherals and manages the attachable of those devices to the system, the fact that we have now moved the actual Tape Control and Printer Control systems outside the kernel would indicate that at least part of this subsystem could also be moved out.

Finally, almost no analysis was made on the potential size and complexity reductions that could be accomplished by replacing the present very sophisticated resource management algorithms (page removal, working set management, multiprogramming scheduling, disk queue management, disk track assignment, and supervisor table management) with simpler versions. Any such proposed change, for creditability, would have to be accompanied by a trial implementation and extensive benchmark performance testing, so as to understand the performance cost of relying on simpler algorithms.

Applicability to Standard Multics

Although the assumed goal in these simplifications has been to reduce the size of the Multics kernel, the application of these proposals to the standard Multics system would also have benefits. In many cases, the simplifications employed in trial implementations exposed bugs and/or security flaws in the standard Multics system. This seems to be a general rule: The simpler, more straight-forward a system is, the better it is understood, and, therefore, the less likely it is to have bugs.

Implementation of Secure Multics

Since this 38,000 line figure is an upper-bound on the size of the secure Multics kernel, we can make some estimates about the amount of manpower required to implement it. Computer folklore tells us that programmers can write about 200 lines of well-debugged code per month. This means that to implement the secure Multics kernel should take about 190 man-months -- about 8 people working for about 2 years.

This number does not include the programming of the non-kernel portions of the system. It represents the basic cost of implementing a secure version of a Multics-like operating system on an arbitrary hardware base. The non-kernel portions of such an operating system are likely to be highly machine-independent and written in higher level languages. Thus, they should be transportable from one system to another.

This empty page was substituted for a blank page in the original document.

PART IV. CONCLUSIONS AND RECOMMENDATIONS

This research project has demonstrated conclusively that if the goal is to simplify and improve auditability, substantial reduction in the size of the security kernel (in comparison with a system not explicitly designed that way) can be accomplished without damaging either the performance or function of a Multics-class operating system. This demonstration is, we feel, quite encouraging to proponents of the security kernel concept and to the goal of developing future acceptably secure operating systems. We further believe that the last suggested figure of 38,000 lines for a Multics kernel really represents an upper boundary on the necessary size. One would expect that if a designer sat down with the radically reduced set of functions represented by that last round of evaluationary changes, and systematically developed a new design from the ground up but to the same specifications, that this new design, being less constrained by history, should be simpler, smaller, and perhaps even a better performer. Current projects to make a security kernel for the UNIX system on the PDP-11 computer suggest that a lower boundary based on less ambitious functions is near 4000 lines. Thus, these two projects provide an order of magnitude target within which new operating system kernel projects should expect to land.

The primary piece of further work that we would recommend would be to carry out that new ground up design, to see how it ends up, and also to carry forward into experimental trials both design-to-model verification and implementation-to-design verification for systems at this level of complexity. Only with these two further steps can general-purpose, secure systems ever be expected to become available.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 10 / 26 / 95

Report # LCS-TR-196

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 112 (119-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☐ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☐ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): FOLLOW TITLE, FORWARD, ABSTRACT, PAGES 45, 47, 71, 109, 111.

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-112) UNNUMBERED TITLE, BLANK, FORWARD, BLANK, ABSTRACT, BLANK, 7-45, UNNUMBERED BLK, 47, UNNUMBERED BLK, 49-71, UNNUMBERED BLK, 73-109, UNNUMBERED BLK, 111, UNNUMBERED BLK. (113-119) SCANCONTROL, COVER, SPINE, PRINTER'S NOTES, TRGTS (3)

Scanning Agent Signoff:

Date Received: 10/26/95 Date Scanned: 11/15/95 Date Returned: 11/16/95

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**