M.I.T. Laboratory for Computer Science
Computer Systems Research Division

June 9, 1977
Request for Comments No. 143

DRAFT OF FINAL KERNEL DESIGN TASK REPORT

by David D. Clark

We are now preparing a final report on the kernel design project, which will be submitted to Honeywell and published as a technical report. Part of this report is a summary of all tasks that were a part of this project. This RFC is a draft of that summary.

Please read any sections that describe work you did. Let me know if I have misrepresented or omitted any of your results. Also, let me know if I have failed to include any task.

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 RFC 37.

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.

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 published as RFC 5, RFC 46, RFC 47, RFC 59, and RFC 92.

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 by logging in interactive processes 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 precision was not required, since the majority of the projects that we performed were not carried through to an implementation which was sufficiently tuned to yield more than very rough performance information. We did perform a variety of small projects as a part of this task, including 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".

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 algorithm did require supervisor privilege, and there was some fear that this residue would complicate the outright extraction of the algorithm. 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. Thus, 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 algorithms from the kernel might significantly degrade the performance of the system. The conclusions drawn from this project were that the outright removal of certain algorithms 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 enahnced 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, by P. Janson, 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.

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, and 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 increase 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 kenel, 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, which 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 current initialization procedure is 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 means that each program runs in a slightly different environment than its predecessor. It is characterizing this large number of different environments which makes 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 could be tailored to reflect the particular physical configuration of the hardware available for this particular start up of the system. Thus, a single Multics tape containing the initialization programs could be generated which would bring up a running Multics on any configuration, in contrast to other systems which require the generation of a tape specific to a 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, we loaded into memory a bit string which constituted a version of Multics capable of running on any configuration. 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, then it was possible to generate a version of Multics which 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 fleged Multics, rather than run in the environment that they are attempting to create. The second phase of initialization consisted of a series of dynamic reconfigurations which modified the minimal Multics to take advantage of the particular hardware and software available 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. It was necessary to demonstrate that certain supervisor tables, such as the traffic control and segment management data bases, could be grown, and implementations were 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 which is 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 structure, in certain cases, has the advantage that it isolates as a sequential process an algorithm which by its nature wants to be sequential, and is 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 algorithm, commonly called page control. When a user references a page not in core, 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 strong 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 the problem of 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 one is not 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 an assembly language program 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 was 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 these algorithms now isolated in separate processes, and the higher levels of the supervisor still running in user processes. In fact, these connections between the core removal process and the higher levels of the supervisor turn out to be some of the stickiest problems associated with this version of the algorithm. The problem is that higher level algorithms 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 core 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 RFC 135 by R. Mabee.

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 which responds to an interrupt runs in a very unusual and limited environment, with restrictions such as it cannot call a locking primitive or perform any other action which might conceivable abandon 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. Although we performed no implementation for interrupt handlers other than the teletype handler, we believe that the use of processes in this way is generally applicable, since the typewriter handler is a very complex piece of code which demonstrates most of the problems relevant to interrupt handlers. 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.

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, being translated into a special call to the traffic controller, which 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. We explored the proper structure of the user computation as a number of processes, and produced a running implementation, although the results of this research were never published as an RFC. 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 no more information than necessary is lost. If an error is not so severe, it may still be necessary to reflect the occurrence of this error to some module other than the module which actually discovers 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 in violation of the layering strategy. During the course of this project we performed a variety of studies to try to understand how Multics should recover from errors, and whether steps taken to insure the reliable recovery of 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 algorithms could be structured in such a way that only a small portion of the algorithms 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, but which at the same time 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 algorithm 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 concept of type extension we proposed as the basis for organizing a virtual memory mechanism. A virtual memory mechanism should be regarded as implementing abstract information containers (e.g. segments) out of physical information containers (e.g. core blocks and disk records). Further, we showed how one could implement the programs and the address space of the mechanicm 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 which 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 memory the pages that 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 must be readied for execution whose state is known only at the higher level. 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.

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 which 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 which 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 provided the interface to the yet higher layers. The second layer provided an intermediate abstraction, which 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 can not 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 must be 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 which takes 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 parallel process whose behavior in some sense drives the kernel modules managing the connection. This differing structure provides 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 measure. 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.