M. I. T. Laboratory for Computer Science

Computer Systems Research Division

August 5, 1976

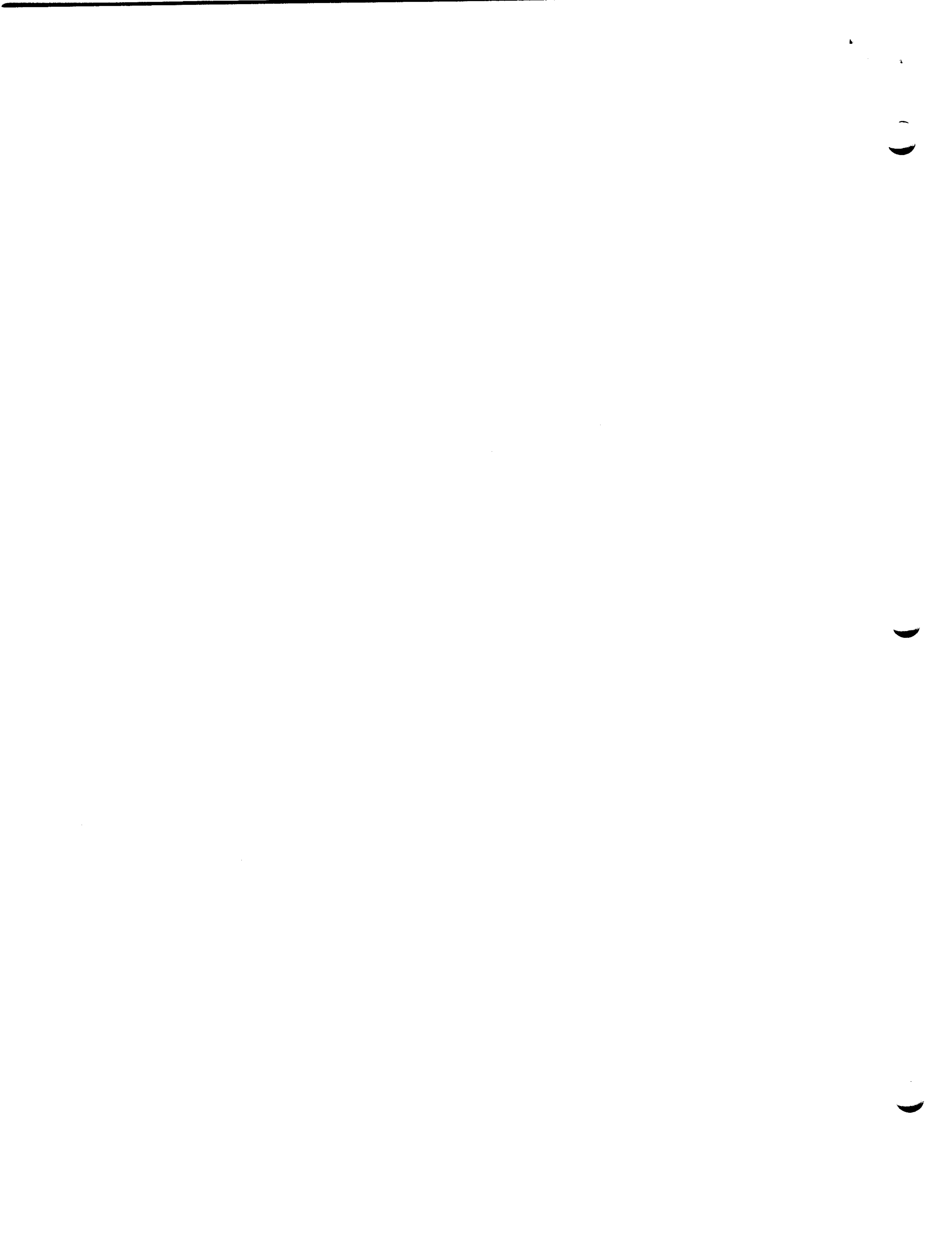
Request for Comments No. 120

A Two Level Virtual Memory Manager

by Andrew H. Mason

Attached is my recently accepted Master's thesis proposal.

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



Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge, Massachusetts

Proposal for Thesis Research in Partial Fulfillment

of the Requirements for the Degree of

Master of Science

Title: A Two Level Virtual Memory Manager

Submitted by: Andrew H. Mason
50 Townsend Rd.
Belmont, Massachusetts
02178

Signature of Author

Date of Submission: August 2, 1976

Expected Date of Completion: January, 1977

Brief Statement of the Problem:

In computer systems, excessive complexity can make the system difficult to understand in detail. The Multics virtual memory manager suffers from this defect. It coordinates the usage of the physical and logical memory resources. The page management module, or page control, is one part of memory management. It is responsible for implementing the demand paging algorithm. Segment control, another part of memory management, supports the segmentation feature of the virtual memory. In the current system, the two parts are closely intertwined. Each depends on the other, making both more complex and harder to understand. This paper examines the nature of the dependencies and proposes study into alternative implementations of the memory manager.

Supervisor Agreement:

The program outlined in this proposal is adequate for a Master's thesis. The supplies and facilities required are available, and I am willing to supervise the research and evaluate the thesis report.

David D. Clark, Research Associate in
Electrical Engineering and Computer Science

Introduction

Time-shared computer systems are being used for more and more applications every day. As they become more wide-spread, the need to be able to guarantee certain properties about them is also growing. Consequently, techniques of proving or certifying operating systems are being studied [3].

In order to facilitate certification, the operating system should be made as simple in structure as possible, yet, at the same time, support a sophisticated functionality. While there are no hard criteria for evaluating the simplicity of an operating system, one ad hoc guideline is that a competent programmer should be able to understand how any module in the system works after studying the code for a few hours.

Since an operating system is comprised of many modules, the task of simplification must proceed slowly, module by module. For my Master's Thesis, I propose to examine various ways to simplify the operating system module known as _virtual memory management_ in the context of a large, segmented memory [1]. The result of the thesis should be the design of a simple but sophisticated virtual memory manager for the Multics system.

Virtual memory management encompasses two distinct functions. First, it must provide facilities for translating a two-dimensional virtual address into a physical address usable by the hardware. This must be accompanied by the ancillary capabilities of adding and removing segments to the virtual memory. In other words, the manager must be responsible for mapping the

two-dimensional virtual memory into the set of one-dimensional memory devices available to the system. Second, the manager must organize the system's physical memory resources in such a way that a user of the virtual memory need not worry about the location of his files. This means that if a user wants to reference a word of the virtual memory that is not currently hardware addressible, the virtual memory manager must move the word to an addressible place without the user's explicit request. One common algorithm used to implement this function is known as demand paging.

By its very nature, memory is important to a computer user. Therefore, a user must be able to trust in the correct operation of the virtual memory. Furthermore, virtual memory is easy to use, so other supervisor modules are implemented using it, rather than trying to handle the complexities of physical memory. Thus, the correct operation of the virtual memory has a strong impact on the correct operation of the entire system. This motivates study into ways of simplifying, structuring, and certifying the virtual memory manager.

The context of the thesis will be the Multics system [4], which is marketed by Honeywell Information Systems, Inc. Multics was chosen for four reasons: First, Multics is a real, live operating system serving real users that has the requisite feature of a segmented, virtual memory. Second, one of the current activities of the Computer Systems Research division is to study better ways to engineer the Multics security kernel. This thesis is directly related to that project. Third, Honeywell is in the final stages of

implementing a new version of the Multics storage system, so ways to improve the virtual memory manager are pertinent to their effort. Finally, several Multics installations are easily available, both directly and over the ARPA network.

Current Structure of the Multics Virtual Memory Manager

The Multics virtual memory manager is made up of two main modules. The first, known as segment control, performs the translation function from a virtual address to a physical device address. Virtual addresses have two components: a segment number and an offset. The segment number is a per-process unique number that identifies the segment to be referenced. The offset indicates the proper word within the segment. For ease of physical device management, segments are broken up into 1024-word pieces called pages. Since all pages of a segment must be permanently stored on the same physical secondary storage device, a physical device address is composed of a (per-segment) physical device identifier, a page address on the device, and a word offset within the page.

Segment control manages the mapping by maintaining information on every existing segment. Some of this information is called a file map. The file map is a table which contains one entry for each page of the segment. The entry describes the physical location of the page. Other kinds of information kept about each segment include, for example, the segment's current length in pages and the date and time that the segment was last modified. Naturally, the volume of this information for all segments is quite large and cannot all

fit into primary memory at the same time. Therefore, if a segment is in use, it is called an active segment and information about it is kept in a primary memory data base called the Active Segment Table (AST). Information about inactive segments is kept in secondary storage.

The second module of the Multics virtual memory manager is called page control. It is responsible for moving pages among the secondary storage devices, primary memory, and the paging device (1) to satisfy memory references. When an attempt is made to reference a page that is not currently in primary memory, a processor exception called a page fault occurs. The process that took the fault enters the page control part of the supervisor and tries to bring in the page. If page control is successful, the instruction that faulted is restarted and the process continues from where it was interrupted.

In addition, page control handles allocation of new pages to segments. Conceptually, every segment contains 256 pages at all times. If a page has never been written, it is defined to contain zeroes. With certain exceptions, only those pages that are non-zero are physically kept in secondary storage. For page control, this means that when a reference is made to a zero page, it must make the appropriate resource control checks, create the page, and assign it to a secondary storage location.

---

(1) The paging device serves as an intermediate holding station for pages. It is larger and slower than primary memory but smaller and faster than secondary memory. Currently it is made out of slow bulk core.

Complexities in the Virtual Memory Manager

The reason that the virtual memory manager is complex and hard to understand is that there is no consistent, well-designed model on which the implementation is based. Instead, assignment of a particular function to a particular module is done on a case-by-case basis by the implementer. For the most part, the system has evolved by adding features to the implementation rather than by developing a new model which incorporates the new feature.

Consider, for example, the quota problem. Quota refers to the number of secondary storage records that may be allocated to (used by) the collection of segments in the sub-tree under a given directory. When page control creates a new page, it must authorize the usage of an extra secondary storage record. Storage resources are allocated on a per-directory basis and are kept in the AST entry for some other active segment (a parent directory). The problem is in designing a resource authorization mechanism that does not violate the modularity of the system.

In order for page control to authorize resources in the current system, page control follows a series of indirect pointers to the proper quota cell. A quota cell is part of an AST entry and contains the needed accounting information for the allocation of storage records. Based on the contents of the quota cell, page control decides whether the resources may be allocated, and, if permissible, allocates them. This mechanism is flawed because segment control is responsible for maintaining the indirect pointers. Therefore, the proper operation of page control, as currently defined, depends upon the

proper maintenance of indirect pointers by segment control.

The central issue behind this upward dependency is at what level in the virtual memory manager should resource control be performed? If page control is made responsible, the implementation must work to avoid upward dependencies. If segment control is made responsible, it becomes difficult to process the creation of new pages in an efficient manner. This problem exists because quota management was incorporated into the virtual memory manager without redesigning the model on which memory management was based.

One of the manifestations of this mutual dependency is a phenomenon called data base communication. The modules page control and semgent control are allowed to communicate to each other through a mutually shared data base (the AST). If, instead, communications were only allowed through well-defined entry points, mutual dependencies such as the one above would be much harder to create and maintain.

Another consequence of complexity is the proliferation of different types of segments in the supervisor. At the present time, the supervisor makes use of each different type of segment for a different purpose. For example, page control is implemented in unpaged segments, while segment control is paged, but always active. The pages of Input / Output buffers have fixed absolute addresses in primary memory, but some of these buffers are not even in the supervisor at all! (i.e. not in ring zero) Although each type of segment exists for a good reason, the fact that there are so many types of segments

strongly suggests the lack of a consistent overview in the virtual memory manager.

Applicable Techniques to Reduce Complexity

Five techniques of program organization seem useful to help structure the virtual memory manager. The first is called the "principle of least privilege" [7]. The central idea is that a program should execute in the environment which embodies exactly those capabilities and privileges that the program needs to operate correctly. If the environment has too few capabilities, the program cannot operate at all. If the environment has too many, there is a risk that the program might abuse or might be made to abuse some of its unneeded privilege.

The second is known as the technique of "least common mechanism" [6]. If two or more programs share a subroutine, the subroutine should only perform those functions needed by all users. If the subroutine does not perform these functions, then the same function is being performed in more than one place. On the other hand, if the subroutine performs some functions for all users and, at the same time, some added functions for only some users, it is doing unnecessary extra work and is more complex that it has to be.

The third technique presents the concept of "layers of abstraction", originally propounded by Dijkstra [2]. Here, the module should be broken up into layers, each implementing a specific, well-defined model. Higher layers may use lower layers, forming a lattice of dependency among the layers. Lower

layers, however, may not use or depend on higher layers in any way. Since each layer corresponds to a model, the task of verifying that the module preserves the properties of the overall model becomes fairly simple. One need only check the correspondence between each model and the appropriate layer, and the check that the layers use each other properly.

The fourth, described by Wulf [8], stresses the importance of separation of policy from mechanism in a module. Those programs necessary to implement the algorithms used by a module (mechanism) should be segregated from those which provide extra features to module users (policy). Unfortunately, this separation is rarely clear-cut, but the division has two attractive properties. First, the policy program can be implemented using the mechanism. If the mechanism provides some desirable feature (such as virtual memory), this can be very helpful. Second, properties of the mechansim can be certified without having to also certify the policy.

The final technique is the rule of "hiding of information" due to Parnas [5]. This rule requires that all interactions among modules occur at the interfaces and, therefore, that the interfaces should be as simple as possible. Internal algorithms and databases should be invisible to all users of a module. This seems especially applicable to operating systems software because one module might have to contend with many different implementations of another module as the system evolves.

All of these techniques stress the same theme: necessary, primitive operations should be separated from optional, usage-oriented operations and implemented at a lower level. However, these techniques also require a model of the system to determine exactly which operations are primitive. The appropriate model will be one of the results of this thesis.

Phil Janson, in his Doctoral Thesis, presents a structure, called a _software cache_, which is very useful for modeling memory systems. Briefly, the structure takes a data abstraction point of view. Janson noticed that sometimes, two data abstractions contain the same kinds of information. One of them supports many operations and is called the _cache container manager_. The other, called the _slow container manager_, only supports read and write operations which transfer the contents of a slow container to or from a cache container. On top of these, Janson constructs an _abstract container manager_ which supports the operations of the cache container manager on all of the containers available to both managers.

Although there are several variations on the basic structure, it fits much of the Multics virtual memory manager nicely and naturally. I expect that Janson's structure will be useful in the modeling phase of the thesis.

Plan of Research

The overall goal of this thesis will be to redesign the Multics virtual memory manager. The new design must preserve the functionality of the current design, yet be simpler and easier to understand. The specific focus of the

new design will be the separation of segment control and page control. At the same time, I hope to solve the problems brought out in this proposal.

The research will be carried out in three phases. First, I will generate a detailed model of my simplified virtual memory manager. To do this, I will attempt to fit the current system into a model based on Janson's software cache structure. From there, I will develop the new model. This phase should take about one month.

The second phase will be the new design. Based on the results of the first phase, I will design the new virtual memory manager. It will be simpler than, but functionally equivalent to, the current one. This phase will take between one month and one and one half months.

In the third phase, I will attempt to show that my design is, in fact, workable. While no full implementation is planned, I expect to implement enough of my ideas so that my design will be convincingly demonstrated. Two months will be needed for this phase.

Overall, this thesis should take no more than five months. In addition, I will need computer time on a Multics installation for coding and debugging the implementation.

References

[1] Dennis, J. B., "Segmentation and the Design of Multiprogrammed Computer Systems," IEEE International Convention Record, Institute of Electrical and Electronic Engineers, New York, 1965, Part 3, pp. 214 - 225.

[2] Dijkstra, E. W., "The Structure of the 'THE'-Multiprogramming System," CACM, Vol. 11, No. 5, May 1968, pp. 341 - 346.

[3] Neumann, P. G. et al., A Provably Secure Operating System, Stanford Research Institute, Menlo Park, Calif., June 1975.

[4] Organick, E. I., The Multics System: An Examination of its Structure, MIT Press, Cambridge, Mass., 1972.

[5] Parnas, D. L., "A Technique for Software Module Specification with Examples," CACM, Vol. 15, No. 5, May 1972, pp. 330 - 336.

[6] Popek, G., "A Principle of Kernel Design," AFIPS Conf. Proc. 43, pp. 977 - 978, NCC 1974.

[7] Saltzer, J., "Protection and the Control of Information Sharing in Multics," CACM, Vol. 17, No. 7, July 1974, pp. 288 - 402.

[8] Wulf, W., et al., "Hydra: The Kernel of a Multiprocessor Operating System," CACM, Vol. 17, No. 6, June 1974, pp. 337 - 345.