UNIVERSITY OF CALIFORNIA

Los Angeles

The UCLA Security Kernel

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in Computer Science

by

Evelyn Jane Walton

1975

The thesis of Evelyn Jane Walton is approved.

David F. Martin

Richard R. Muntz

Gerald J. Popek, Committee Chairman

University of California, Los Angeles

1975

To my mother, without whose help and support this work would never have been possible,

and

to my cats:

S.I.D., Duncan, and Maggie

who did their best to prevent its completion.

Table of Contents

Acknowledgements

The design of the system described in the following sections is not solely the work of the author. It represents the outcome of several months of serious design work undertaken at UCLA under the direction of Professor Gerald J. Popek. The initial design work was accomplished through the joint efforts of Professor Popek, Charles S. Kline, a Ph.D. candidate under Professor Popek, and the author. Revisions to the initial design were made by Professor Popek, Charles Kline, Steven Abraham, and the author.

It is extremely difficult to succinctly separate the contributions made by the author from those of the other participants, and no real attempt has been made to do so in the text of the thesis itself. However, some general comments regarding the division of credit (or blame, as the case may be) can be made: 1) the entire general system design presented in the initial sections of the thesis was accomplished through the joint effort of the previously mentioned individuals, although the description of this design is solely the work of the author; 2) the design of the kernel calls presented in later sections was in large part accomplished jointly; the descriptions of each call, however, are again solely the work of the author; 3) the input/output specifications and any notation used in them

List of Figures

Descriptions of kernel primitives are made, along with the problems and constraints which provided the motivation for their final format. Input/output specifications for the actions required of each kernel primitive -- in a style similar to that developed by Parnas -- are given.

Introduction

This thesis describes the design of the kernel of the UCLA Virtual Machine System. The thesis is organized into several parts. Early sections are in many respects independent of all others, but later sections tend to expand, amplify, and otherwise build upon information and materials presented in preceding sections. Sections are ordered with respect to increasing level of detail, such that early sections present overviews and generalities, while later sections are concerned with the more detailed aspects of design and implementation.

Chapter 1 presents an introduction to the problem and a number of the basic design decisions which influenced the overall structure of the system. Chapter 2 describes the hardware and software components of the system. Chapter 3 discusses a number of important kernel design decisions, their impact upon the system as a whole, and the general organization of the kernel itself. Chapter 4 describes the set of primitive operations supported by the kernel. Chapter 5 discusses the interesting design issues related to the kernel primitives described in chapter 4.

ABSTRACT OF THE THESIS

The UCLA Security Kernel

by

Evelyn Jane Walton
Master of Science in Computer Science
University of California, Los Angeles, 1975
Professor Gerald J. Popek, Chairman

This thesis describes the design of the security kernel of the UCLA Virtual Machine System. The motivations for the construction of a secure computer operating system are sketched, the advantages which a security kernel approach provide are given, and the utility of a verifiable security system is discussed. Simplifications to the security kernel approach provided by the selection of a virtual machine system as its underlying target software are described.

System design goals are presented and the impacts of these design goals upon the design of the system examined. Detailed descriptions of the major design decisions are given. The interface between the kernel and other system components is presented.

are solely of the author's devising; and, 4) although the general design of most of the kernel calls was indeed a joint effort, the examination of many of the alternative solutions to the problems presented and their analysis are for the most part entirely the work of the author.

The author wishes to thank Charles Kline and Gene Schneider for their reading and helpful comments on early versions of this document. The author is especially indebted to Steven Abraham for his multiple readings and comments made on several versions of this document and especially for the aid he rendered in detecting and removing various inconsistencies throughout.

Last, but not least, I wish to thank Professor Popek, my committee chairman, for the great patience he showed through multiple readings of disorganized drafts and for his sympathetic, yet critical suggestions. Without his support and constant prodding this document would never have been completed.

This research was supported in part by Advanced Research Projects Agency, Computer Network Research Contract No. DAHC-15-73-C-0368.

Chapter 1: Basic System Design Decisions

1.1 Problem Statement

As the usage of computers in normal, everyday life has increased, the need for reliable computer software has grown. At the same time, the desire to share computing resources has become intensified. The expanded capability for successful computer resource sharing has in turn increased computer usage in the business field. A company which once could not afford to automate its billing procedures-- for the simple reason that it could not afford to purchase and maintain its own computer facility -- is no longer quite so constrained: it is now possible to lease or buy time from computer firms, thus gaining computational power without also gaining the expense of personally maintaining the entire computer facility as well.

As more and more businesses rent computer time and share their computational resources with other users, possibly direct competitors, the need for some guarantee of computational security grows. If important facts and figures are kept in a computational data base, figures vital to the corporate existence of the firm in question, then it is of extreme importance that those facts and figures <u>not</u> become available to persons who might use them in a manner detrimental to the corporation. Some assurance

is needed that the computer which is to be trusted with sensitive information can indeed be trusted to deal with that information in a manner commensurate with its sensitivity. Some assurance is necessary regarding the the security of the computer system.

Thus the motivation for the existence of secure computer operating systems arises. Currently, "no protection system implementation of any major multi-user computer system is known to have withstood serious attempts at circumvention by determined and skilled users".[POPEK74A] Although serious effort has been given to the task of adding (retrofitting) security features to existing computer systems, it seems that this task can well become a self-perpetuating one-- a matter of first discovering security flaws, then correcting them (perhaps introducing new ones in the process), and finally starting over again, without ever having any real assurance that the final flaw has been eliminated. [1]

One major problem in attempting to retrofit security is that often one is attempting to impose a coherent structure upon an entity which has little coherent

---

[1]. As Dijkstra has stated, "Program testing can be used to determine the presence of bugs, never their absence", an observation true of computer programs in general and equally as true of operating systems (which are, after all, only a special type of computer program).

structure in the first place: it is difficult to add security features to a system structure designed without security in mind. One suspects, then, that an approach which considers security issues from the very beginning--indeed builds the entire system framework around security considerations (instead of vice-versa)-- would have a higher probability of success as far as achieving reliable security is concerned. An additional factor one would hope, would be that advances in computer architecture might also simplify the solution of certain problems which arise from security considerations. [2]

It should be easier to design a secure system from scratch than to secure an existing operating system which was not designed with security considerations in mind. It seems fairly clear, however, that just building a system with security in mind does not guarantee the security of the resulting system any more than building a system without considering security does. [3] One might claim security and perhaps have personal reasons for believing one's system secure after designing it to be so, but merely

---

[2]. This line of reasoning could also be used as an argument for not building such a system at the present time: why use possibly inferior materials if better materials may be shortly forth-coming? One problem, of course, is that better materials will probably be developed only when sufficient demand for them is generated. A second problem is that it is difficult to decide how to build a better mousetrap if one has no real idea what is wrong with one's current mousetrap.

claiming security is not sufficient. One would wish some way of demonstrating security and demonstrating it in a positive sense: One would like to be able to say "The system is secure", not merely a negative "No one has discovered any flaws yet."

The question arises then of how such positive assurance of security can be obtained. A partial answer has been advanced which requires two specific actions be taken: 1) a rigorous working definition of security must be formulated and translated (if necessary) into terms which are concrete enough to be dealt with in a precise manner; and 2) a formal verification of the operating system must be made in order to determine if the system enforces security as it has been defined in 1).

If these steps are taken and the proof advanced by 2) subjected to intense scrutiny and subsequently determined to be correct, then a convincing demonstration of the security of the system relative to the definition supplied will have been accomplished. [4] Accordingly, the

---

[3]. This is not entirely true, of course. One suspects that some level of increased security has been obtained. The point here is that just "thinking" about security provides no concrete assurance of its existence.

[4]. This demonstration is not really one of correctness, but rather one of consistency; the system is consistent with the security definition.

decision was made to design the system from scratch with the end goal of formally verifying its security properties.

The first step in this process was to formulate a set of initial design goals. These goals were:

1. Verification - it should be possible to formally verify the security properties of the resulting system.

2. Usefulness - the resulting system should be a useful system.

3. Efficiency - the resulting system should be relatively efficient, in order to demonstrate that security can be cost-effective.

4. Cost - the system should be relatively inexpensive to build, in terms of both time and money.

1.2 Basic Design Decisions

Although most of the design decisions regarding the structure of the system were made after a great deal of careful consideration and weighing of alternatives, some decisions were fixed from the start. Often one is forced to work with the materials at hand. Such was the case here in choosing the target machine for the security system: a Digital Equipment Corporation (DEC) PDP 11 computer was already destined to be purchased in order to provide network access for UCLA ARPA contract personnel and would be the most accessible local machine for system construction; hence this machine became the target computer for construction of the security system. [5]

Two other structural decisions occurred implicitly without much discussion at all. First, was the decision to build a multiprocessing system. This was a natural outcome of the observation that one of the primary usefulness of any security guarantee is that it proports to protect one user from another, a protection that seems much less useful if only one user can access the system at a time.

---

[5]. This is not strictly true. It was not merely a matter of "take what you can get". Originally a PDP 11/40 was to be purchased, but instead an 11/45 was purchased and certain hardware modifications made to the 11/45 because it was to be the target machine of the security system.

The second implicit decision was that the system was to be primarily an interactively oriented one. The real justification for this decision is that the interest and experience of those involved in the system design has been mainly in the area of interactive, time-sharing systems.

1.3 Security

With the primary design goals firmly in mind, the design of the UCLA VM system began. Since the primary goal was a system that offered verified security , it was first necessary to formulate a workable definition of security relative to the design of the system. Two varieties of security were considered. The first, which has been termed data security, is concerned with the privacy of data and the controlled access to it, and has been defined as the protection of data (files, segments, etc.) from accidental or intentional disclosure to unauthorized persons and from unauthorized modification. [IBM1]

The second category of security is concerned with the issues of productivity and denial of service. In addressing this facet of the security issue one typically seeks to give assurance that the performance of a given process cannot be substantially degraded or altered by the actions of other processes in the system, or, perhaps more importantly, that the actions of one process can not cause another process to malfunction in either predictable or unpredictable ways.

The design of the UCLA virtual machine system was focussed mainly on problems in the area of data security. Although productivity and viability are important issues in

their own right, they can be tackled much more effectively once the problems of data security have been solved. Once data security has been reliably enforced, one suspects that the elimination of productivity-related flaws will be greatly simplified. Issues of productivity and viability have not been ignored completely, however. The constraint that the system be a useful one implies that some viability and productivity issues must be faced and dealt with.

Issues of data security can be further subdivided into areas of policy and enforcement. The policy area is concerned with questions regarding who should be allowed to access what information, in what manner, and under what circumstances. Examples of policy issues occur frequently in the military environment, where decisions of who should be granted what level security clearance must be made, and where questions relating to the security classification of important documents must be answered. [6]

The second area, enforcement, is concerned with guaranteeing that policy decisions are enforced once they have been made. In a military environment, this task might include insuring that an employee with top-secret

---

[6]. Questions such as: Should employee X be given top-secret security clearance? or, What security classification is appropriate for document Y?

classification and access to top-secret documents does not pass on top-secret information to another employee who possesses only a secret clearance, or the more general task of protecting classified documents from pilferage by unauthorized persons.

The primary focus of attention in the UCLA system is upon the enforcement aspects of data security rather than the policy aspects. A large class of "reasonable" security policies can be implemented (and hence enforced) by the system. [7] Hence concern is given mainly to the enforcement aspects of a security policy and the means by which security enforcement can be convincingly demonstrated.

---

[7]. The class of security policies implementable under our system is not as large as the class of policies implementable by some other security systems (Hydra, for instance, [WULF74]). Our security is not extensible in the sense that we have not allowed new object types or access capabilities to be defined.

1.4 Verification

It is our contention that any convincing demonstration of security enforcement must be made upon some firm theoretical basis, rooted in the modelling of those aspects of security enforcement relevant to computer operating systems and in their subsequent implementation in such a manner as to make the correctness question decidable through the methods of formal program verification.

What influence has the verification goal had upon the design of the system? What constraints, if any, has verification placed upon the structure of the system?

The first and probably foremost constraint is upon the size of the system. Unfortunately, the task of verifying even the simplest of programs is a non-trivial proposition. The largest known program which has been claimed to have been verified is on the order of 2000 lines of high level code.[RAGLAN73] [8] If the system is to be ultimately verified, either formally or informally, using current verification techniques, then it is imperative that the total amount of code to be verified be as small as

---

[8]. The program being referenced here is the Nucleus verification condition generator. This program was, however, too large to be compiled and consequently has never been run. Neither has its proof been widely circulated.

possible.

Secondly, the code should be well-structured. [9] Although unstructured code can be proven correct, it is in general significantly harder than proving well-structured code: A program with purely sequential statement flow is much easier to verify than one which makes unrestricted use of jumps. However, understandability should not be sacrificed merely in order to use a well-structured construct.

Third, efficiency considerations should be secondary to understandability: An algorithm which accomplishes its purpose quickly and efficiently, but at the cost of utilizing all of the strange anomalies of the hardware is better replaced by one which is perhaps less efficient, but more understandable.

---

[9]. The computing community thus far has been unable to precisely define "well-structured" in any rigorous manner, although most people probably have a general notion of what is well-structured and what is not. Perhaps the best definition which can be offered at this time, although one which is not especially helpful from a theoretical point of view, is one which maintains that anything is well-structured if it is easily and conveniently handled by current heuristic and axiomatic verification techniques.

generally found in the usual operating system.

Since the security kernel will be significantly smaller than the usual operating system, its maintenance should be proportionately simplified, and, because it is small, the goal of verification, in the style of formal program verification, should be a feasible one.

1.6 Virtual Machines

The structure of the system thus far presented was one with the security kernel running on the bare hardware and all other software layered around that security kernel. The next question of interest was: what sort of software was appropriate to layer around the security kernel? The second goal, that of a useful system, might have prompted the desire to layer a full-blown, general-purpose operating system over the security kernel. The cost of such a venture, however, was rather discouraging and was in direct conflict with the third goal, the requirement that the system be relatively inexpensive to construct. The solution to this conflict of interests was to build a virtual machine system.

Before defining what a virtual machine system is, it is useful to provide some general background on the utility of virtual machine systems. Most operating systems run on the bare machine and remove some of the normal instructions from the instruction set available to user programs, typically those instructions that change mode and do I/O, and replace them with other macro instructions (macro in the sense of big --not the normal usage of macro in computer contexts) called supervisor calls to accomplish some of those prevented actions in an easier and more

controlled way. In this way the operating system effectively changes the operating environment under which user programs execute. User programs are then written to run in this new environment.

Unfortunately, operating systems vary tremendously and user programs written to run under one operating system on a given machine rarely run under another different operating system, even for the same machine. But suppose one built an operating system which produced an environment identical in most respects to the original machine. [10] Then one would have an operating system for running other operating systems. This is precisely the notion of a virtual machine system.

A virtual machine system, then, is a system for running multiple pieces of computer software in an environment (called a virtual machine or VM) which to the software appears identical to the basic hardware of the computer -- regardless of the environment in which the virtual machine system itself exists. The virtual machine concept is a derivative of the computer simulator concept which allows programs written for a given computer X to be

---

[10]. The environment is not precisely that of the bare hardware. Typically the environment in which the virtual machine operates differs from the read machine with regard to timing characteristics and available resources.

debugged while executing on another computer Y. The major differences between a simulator and a virtual machine system, aside from the fact that a virtual machine system typically produces multiple copies of the machine it provides, are 1) in the case of a simulator, machine X is in general different from machine Y, while in a virtual machine system X and Y are identical, and, 2) performance of virtual machine systems is overwhelmingly better than that of a simulator as a result of the efficiency gained from the fact that the majority of instructions executed for the VM may be executed directly on the basic hardware and do not require software intervention or simulation. The virtual machine monitor acts as a simulator which is simulating the machine upon which it is actually running.

The functions and responsibilities of a virtual machine system can be simply and succinctly stated: "The heart of a VM system is the virtual machine monitor (VMM) software which transforms the single machine interface into the illusion of many. Each of these interfaces (virtual machines) is an efficient replica of the original computer system, complete with all of the processor instructions (i.e., both privileged and non-privileged instructions) and system resources (i.e., memory and I/O devices). By running each operating system on its own virtual machine it becomes possible to run several different operating systems

(privileged software nuclei) concurrently." [GOLDBERG74]

The usefulness of a virtual machine system is not exclusively that of an inexpensive alternative to constructing a full operating system: 1) virtual machine systems allow the convenient modification and testing or new or existing operating system software without seriously impacting the use of the the machine for other daily tasks; 2) virtual machine systems increase the amount and flexibility of existing software available to system users by enabling multiple operating systems and their all-important user software to be simultaneously available; and, 3) virtual machine systems permit test and diagnostic programs to be run while the machine is available to other users, possibly allowing the check-out of previously malfunctioning devices to be made without preempting the entire machine.

There are several advantages to building a virtual machine system that are of particular relevance to the design goals presented earlier: 1) production costs are considerably lower than that of a general-purpose operating system, 2) a number of operating systems already exist for

the PDP 11/45, many of which can be run under the virtual machine monitor [11], 3) virtual machine systems are fairly convenient to use--at least as convenient to use as the virtual machines (operating systems) they run, and, 4) virtual machine system performance is high enough to be considered tolerable.

Instead of constructing a full-blown operating system layered above the security kernel, we chose to build a much smaller (and hence correspondingly less expensive) virtual machine monitor, whose function it is to simulate to user processes those features of the original bare machine which had been altered by the kernel. By choosing a virtual machine approach, we gained the usefulness of a general-purpose operating system (indeed, the usefulness of general-purpose operating systems, since existing operating systems can easily be run under the virtual machine

---

[11]. Not all operating systems for the PDP 11/45 can be run under the current virtual machine monitor, however. There are two basic reasons for this: First, systems which operating in a timing-dependent manner may not function in the same manner as they would upon the bare hardware; second, the memory management unit of the 11/45 has not yet been virtualized. This first problem is inherent in all virtual machine systems. More important for the time being, however, is the fact that the current virtual machine monitor has not yet virtualized the 11/45 memory management hardware, although virtualization of the memory management unit is currently being designed. Until such time as memory management is virtualized, however, no system which makes full use of the 11/45's three state architecture or memory management features can be properly run under the virtual machine monitor.

monitor), while still retaining the lesser construction expense of a virtual machine monitor.

The decision to build a virtual machine system, however, was not one to be made lightly. Although the concepts involved in the notion of a virtual machine system are to a large extent independent of the base hardware machine which is being virtualized, they are not entirely independent: some third generation computers cannot be virtualized -- that is, for some existing third generation computers it is impossible to construct a proper virtual machine monitor. Thus if one wishes to produce a virtual machine system, one must choose the base hardware to be built upon with at least enough care to insure that virtualization is indeed possible.

In [POPEK74B] Popek and Goldberg succinctly state the formal requirements for virtualization of third generation computers, in terms of the relationship between sensitive and privileged instructions: "For any conventional third generation computer, a virtual machine monitor may be constructed if, and only if, the set of sensitive instructions for that computer is a subset of the set of privileged instructions." [12]

Thus some deliberate consideration was necessary to determine whether the target computer, a DEC PDP 11/45, was

indeed virtualizable or capable of being modified in order to make it virtualizable. Such an examination of the 11/45 was made, with favorable results, and modifications to the machine designed in order to make it virtualizable. These hardware modifications are described in a later section.

---

[12]. An instruction is said to be privileged if and only if under all circumstances it traps when executed in user mode and does not trap when executed in supervisor mode. An instruction is sensitive if it attempts to alter the physical resources available or affects the processor mode, or if the effect of its execution depends upon its location in real memory.

1.7 Other Basic System Components

The structure envisioned now consisted of the bare hardware surrounded by a very tiny operating system-like structure, the security kernel. Over the kernel is layered a virtual machine monitor above which virtual machines (operating systems) can be run. This structure is illustrated in figure 1.

At this point in the discussion, two important points remain unresolved: 1) thus far no mention has really been made of how the security policy is represented, maintained, or modified, and 2) no mention has been made of how the kernel associates its protection information with anything so concrete as a user process. The need for addressing these two issues provides the basic motivation for two additional system components: the updater and the initiator.

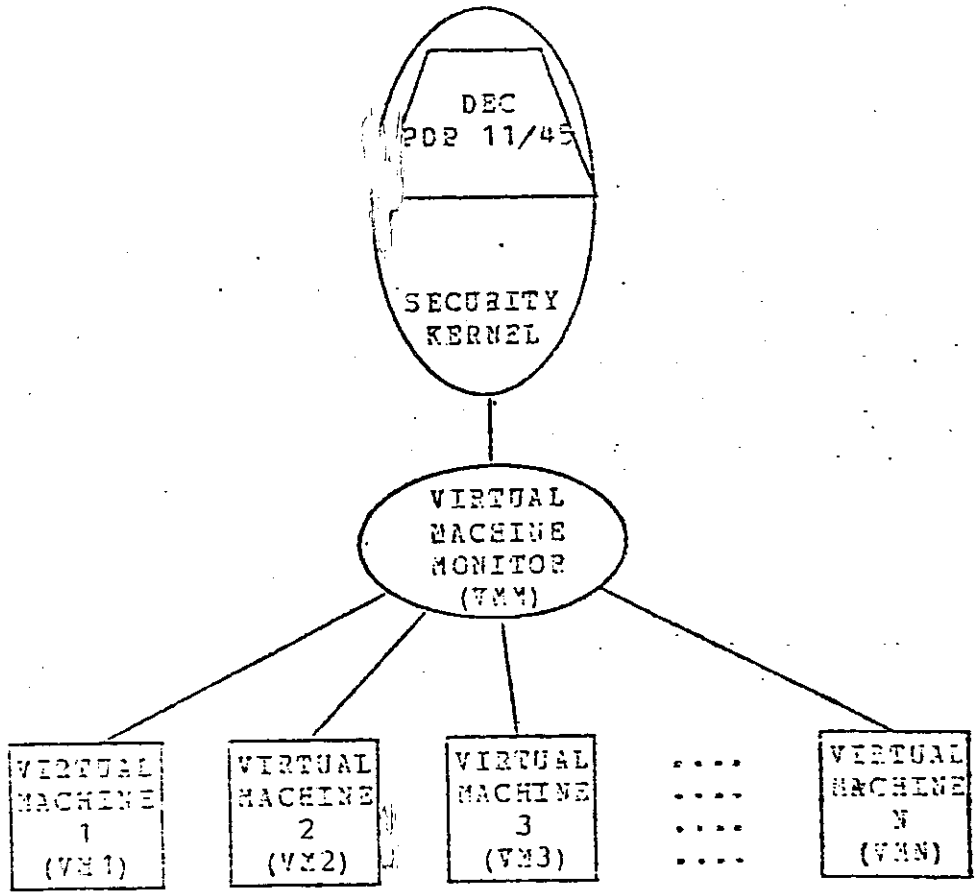

Figure 1: Initial System Structure

1.7.1 The Updater

The updater is the process which controls changes in the kernel's protection data.

The design and implementation of the security policy maintainer, a privileged process called the updater, has perhaps been given less consideration than other components of the system. Such a development is not completely unexpected, however, considering that our main focus lies in the area of security enforcement and not security policy. Since the updater vitally affects the security of the system -- in essence determines most of the meaning of security in the system -- it is necessary that its code be eventually proven correct. [13] To guarantee security enforcement, one must guarantee proper modelling of the security policy, which in turn requires that one demonstrate certain properties about the manner in which the security policy is represented and updated. [14]

---

[13]. It does not, however, determine the entire meaning of security in the system. Policy cannot force the contents of one process's general registers to become visible to another process, for example.

[14]. If one's security policy states that X can access A but that Y cannot access A, then if the updater enters that Y can access A and X cannot access A, then even an otherwise totally correct security enforcer will not have enforced the given security policy.

Although the updater code, like the kernel code, requires verification, it is a readily isolatable piece and one that can easily profit from the protection environment which the kernel provides for processes operating under its control. By layering the updater above the kernel, one is able to utilize the protection facilities already provided by the kernel in the verification of the updater itself. If the kernel can be shown to maintain the data security of processes running beneath it, then this assurance applies also to the updater, and the updater thus need not concern itself with the problems of protecting itself from malevolent processes.

1.7.2 The Initiator

The initiator is the process to which all consoles initially belong. The security-relevant property of the initiator is that it is allowed to identify the user to the kernel in such a manner that the kernel is able to make an association between the process identity and the kernel's protection data.

The initiator is the privileged process assigned the task of identifying users to the kernel. The portion of the system which provides user authentication must likewise be proven correct if the verification of the kernel is to have any real meaning. [15] User authentication methods, like scheduling algorithms, are subject to change as newer (and hopefully more secure) mechanisms and theories are developed. The desire to provide suitably flexible user identification schemes, coupled with the desire to remove as much semantic understanding of I/O operations from the kernel code as possible, as well as the feeling that authentication issues represent a clearly separable task,

---

[15]. One is tempted to question the validity of any claim that user X cannot read user Y's data if all that X has to do to gain access to Y's data is simply to claim that he is user Y. One finds oneself dealing with rather nebulous terms, however, when one speaks of verifying an authentication scheme.

led to the removal of the major responsibility for user identification from the kernel.

The authentication task is, however, a two-sided problem. One needs not only to deal with the problem of proper identification of the user to the system, but also with the proper identification of the system to the user: one would also wish to assure a human sitting down at a terminal that he is typing his super-secret password at the authentication process and not at some other random process masquerading as the authentication process.

A possible solution here would be the development of some type of login protocol. The addition of such a protocol to the kernel code would not only complicate it tremendously code-wise, but would probably also reduce system performance tremendously unless the "protocol" is of an extremely simple sort. The problem of identifying the system to the user is an instance of the "Trojan horse" problem and is discussed at some length in the appendix.

The initiator is responsible for creating and starting all user processes. By conversing with the initiator, a user may specify which new virtual machine he wishes to run or which existing virtual machine (process) he wishes to become attached to.

After completion of preliminary conversation with the user, during which the user has been required to identify himself to the initiator's satisfaction, the initiator identifies the user to the system and either creates and starts the requested process, attaching the console to it, or attaches the user (and console) to an existing process.

1.8 Basic System Structure Summary

The general overall structure of the has now been sketched and is illustrated in figure 3. The system's detailed structure, however, is still quite incomplete at this point, but before the structural details can be presented it is first necessary to examine certain of the security relevant details of both the hardware and software base upon which the UCLA VM system is to be constructed. The examination of these details is the subject of chapter 2.

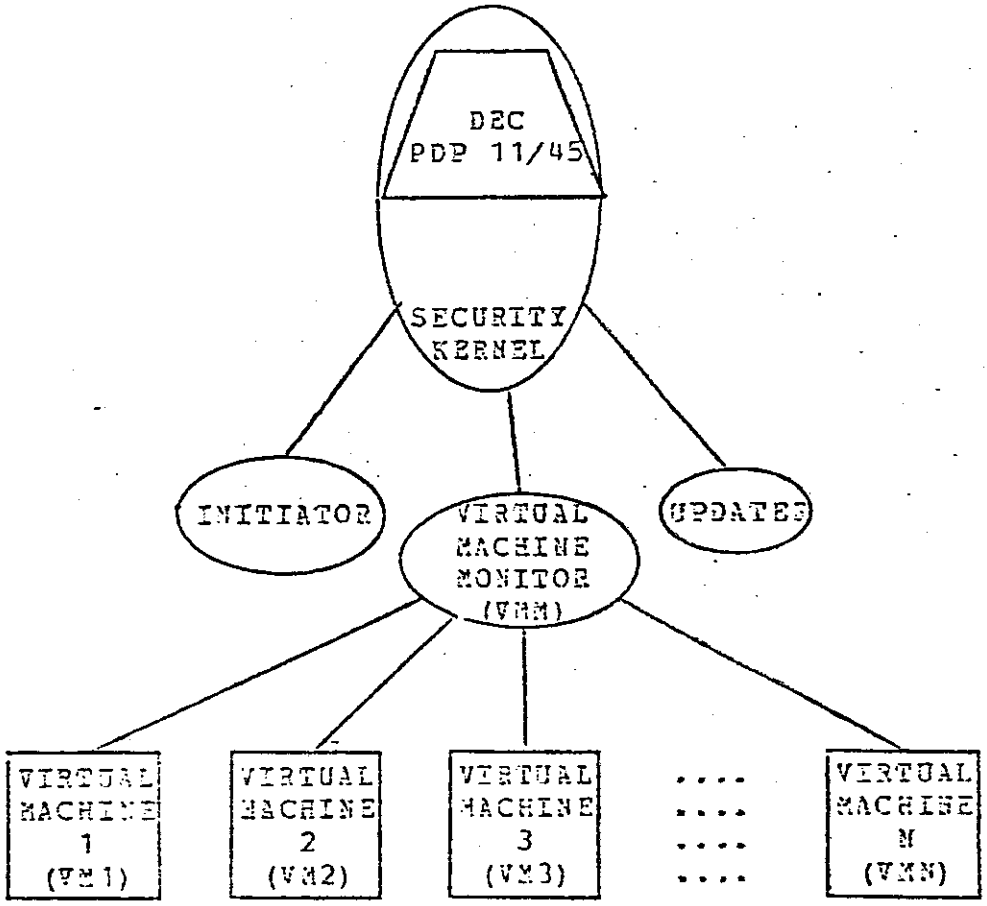

Figure 2: Retiree System Structure

Chapter 2: Hardware and Software Components

2.1 The PDP 11/45 Architecture

This section is intended to be a general overview of the hardware features of the PDP 11/45 which are relevant to protection and security. It is not meant to be a detailed description of the hardware, as a much more accurate description can be found in [DEC73B], and is not really necessary for a reasonable understanding of the remainder of the thesis.

The PDP 11/45 is a small, general purpose, three state computer, capable of directly addressing 28K 16 bit words of memory (without use of memory management) and up to 124K with memory management. [1] It has two independent sets of general registers, each set containing 6 registers. In addition, it provides one special register called the 'stack pointer' (SP) for each program state and a final register, R7, the program counter (PC), which is shared by all programs running in the machine.

Four basic areas of the 11/45's architecture are of interest with regard to security and protection: 1) the

---

[1]. Although the machine can physically address 32K (without memory management) and 128K (with memory management), 4K of these words are effectively lost since the CPU logic automatically maps certain of the upper 4K addresses into device registers.

mode structure of the machine; 2) the memory management unit of the machine; 3) the I/O architecture of the machine; and, 4) the manner in which it handles exceptional (error) conditions.

The PDP 11/45 is a three state machine. Programs operating within the machine can execute in either kernel, supervisor, or user mode (state). Programs operating in user mode are restricted in the kind and number of instructions which may be legally executed. Supervisor mode programs are slightly less restricted in their instruction capabilities and kernel mode programs are unrestricted in their use of the machine. Some of the operations allowed in kernel mode are not allowed in supervisor or user mode and some operations admissible in supervisor mode are inadmissible in user mode. [2]

The current machine operation mode at any given instance is completely determined by the "current mode bits" contained in the processor status word (PS), a

---

[2]. Unlike most machines, however, supervisor mode is only slightly more powerful than user mode. The only major difference is one imposed by the memory management hardware: supervisor mode programs can read and modify both code and data belonging to user mode programs by use of a special group of machine instructions (MFPI, MFPD, MTPI, MTPD). These instructions cannot normally be used by user mode programs to access supervisor mode instructions or data since user mode programs are usually unable to manipulate the previous mode bits which select the virtual space in which these special instructions operate.

special memory cell located in the upper 4K of the machine's memory. The processor status word may be changed in one of three ways: 1) explicitly, by physically modifying it via "normal" machine instructions, just as if it were any other memory word; 2) implicitly, through the use of one of the several machine instructions which cause the PS to be implicitly changed; [3] or, 3) asynchronously, as a result of an interrupt or trap.

The ability to explicitly modify the PS is strictly a function of whether any portion of the program's virtual address space is appropriately mapped into that portion of the machine's physical memory which contains the PS. Programs operating in user or supervisor mode are restricted in the manner in which they may implicitly modify the PS by means of the RTI and RTT instructions in that they are unable to cause the new processor mode to be more privileged than the processor mode at the time the instruction is executed. [4] Asynchronous modification of the PS resulting from an interrupt or trap is totally determined by the contents of the new PS portion of the appropriate interrupt or trap vector, although interrupts

---

[3]. These instructions are the Set Priority Level (SPL), Return from Interrupt (RTI), Return from Trap (RTT), I/O Trap (IOT), Emulator Trap (EMT), Break Point Trap (BPT), and Trap (TRAP) instructions.

are not allowed to set the previous mode bits of the PS. [5]

The hardware mode provides some degree of protection. However this mode protection in itself is not sufficient to provide full protection, for the hardware does not automatically prevent any program from explicitly changing the previous and current mode bits in the hardware processor status word, provided that it has access to the PS, regardless of the program's current mode. Thus a user mode program which possesses write access to the hardware processor status word can put itself into kernel mode simply by clearing (zeroing) the PS and from there do any damage it wishes. [6]

---

[4]. When the RTI and RTT instructions are executed, the new PS is constructed by ORing in the current PS with the current mode bits, previous mode bits, and register set bits of the requested new PS. Since the more privileged modes are numerically smaller (kernel=00, supervisor=01, user=11), this restricts the RTI and RTT operations to making outward calls.

[5]. The true previous mode bits are automatically filled in by the hardware at the time of the interrupt or trap and effectively override the previous mode bits specified in the new PS stored in the trap vector.

[6]. There are certain logistic problems in doing so, however, since changing the current processor mode changes the address space within which instructions are being executed.

The most convenient (and, in fact, the only) mechanism for allowing or restricting access to particular memory locations on the 11/45 is through the use of the memory management unit. The 11/45 memory management unit consists of three sets (one per program mode) of thirty-two 16-bit registers and four status registers. Each set of registers can be divided into two groups, each containing 16 registers: 1) instruction space registers and 2) data space registers. These sets of 16 registers can then be further separated into two additional groups, each containing 8 registers: 1) page descriptor registers (PDR's) and 2) page address registers (PAR'S).

The PAR's describe the physical memory location of the given memory frame, while the PDR's describe certain logical aspects of the frame, such as its length, the "legal" accesses which can be made to it without causing an abort or trap [7], whether the frame has been modified (written in) since the register was last loaded, and the direction in which the frame is expanded. [8]

---

[7]. The difference between "trap" and "abort" is somewhat subtle here. A memory management abort terminates the offending instruction the moment it attempts an invalid access, while a memory management trap allows the instruction to complete before transferring to the memory management trap handler. Memory management "traps" are generally used for gathering memory management statistics, while "aborts" are used to catch page faults and prevent illegal accesses.

Four status registers are associated with the memory management unit. The first status register (SR0) contains various status and error information when memory management faults have occurred. It also controls the enabling/disabling of the memory management unit in its entirety, as well as controlling whether memory management traps will be taken when the conditions for them are met. [9] The second status register (SR1) contains information necessary to recover the original contents of any general registers which may have been modified during the execution of an instruction which resulted in a memory management abort, in order that effective error recovery is possible.

The third status register (SR2) contains the 16-bit virtual program counter which is recorded at the beginning of each instruction fetch or with the trap vector address at the beginning of an interrupt. The fourth status

---

[8]. Memory frames on the 11/45 are allowed to expanded upward or downward. Downward expansion is principally useful in adding additional stack space, since stacks on the 11/45 grown downward.

[9]. It is possible, by disabling the "Enable Memory Management Trap" bit in SR0, to effectively ignore memory management traps. All of the appropriate status information is still recorded, but no actual trap is generated.

Data Space in each of the three program modes. [10]

With Data space enabled, each program mode has the capability of addressing 16 separate memory frames (8 for I-space, 8 for D-space). Each memory frame can range in size from a minimum of 32 words to a maximum of 4K words in 32 word increments (i.e. smallest is 32 words, second smallest is 64 words, etc.). A programs's virtual memory need not be physically contiguous (contiguous in memory) nor logically contiguous [11] and memory frames are not restricted to being of a uniform size.

The distinction between instruction and data space is fairly simple: instructions, immediate operands, index words, and absolute addresses are accessed through the instruction space registers; all other references are made through data space registers (if data space is enabled; otherwise all references occur through instruction space).

When a memory reference is made which results in a memory management trap or abort, the machine takes a memory

---

[10]. Data space can be selectively enabled or disabled for each program mode, if desired. For instance, user mode might have Data space disabled, while kernel and supervisor modes have it enabled.

[11]. It could have holes in it resulting either from "missing" frames or from "short" frames.

management trap to the trap handling routines whose address is given in kernel data space location 250(octal). At this point, the memory management status registers (SR0,SR1,SR2,SR3) become frozen in the state which existed at the time of the generation of the trap. Before exiting, it is the responsibility of this memory management trap handler to re-enable the error checking portion of the memory management unit in order to "unfreeze" the status registers and allow any subsequent memory management faults to be handled properly.

Physically speaking, all memory addresses on the 11/45 are 18-bit addresses. The 11/45 word size, however, is only 16 bits, which restricts the number of specifiable addresses to 32K words. When the memory management unit is disabled, no address relocation occurs, but the extra two bits are automatically generated by the hardware according to the following scheme: the 16-bit virtual address X maps into the 18-bit physical address X' = 00X (unless 160000 <= X <= 177776, in which case X' = 11X [12]), regardless of whether the reference is made in kernel, supervisor, or user mode. Each mode space is protected identically -- that is, not at all -- and all references are made as

---

[12]. This is due to the fact that locations 160000 - 177776 (octal) are automatically mapped into addresses in the uppermost 4K of physical memory when memory management is not enabled, thus allowing kernel, supervisor, and user mode programs access to the device control registers which reside there.

though through instruction space, giving a total address space of 32K. If more than 28K of real memory is available, then locations above the first 28K cannot be immediately accessed by any program running on the CPU itself (since it fetches only 16 bit addresses, effectively) although this area of memory can, of course, be read or modified by most devices operating asynchronously to the CPU (since most devices use 18-bit addresses). Without use of memory management, the only protection is that afforded by the current processor mode, and since all modes share the same address space (and have a priori access to the PS), that is little protection indeed.

Once memory management is enabled, however, an enormous jump in the level of protection occurs, even without making use of some of the more "standard" features of the unit, such as relocation. By merely enabling the trapping mechanism, one can now memory protect portions of the virtual address space shared between kernel, supervisor, and user programs, and, by additionally write-protecting the PS, guarantee that the ability to change processor mode is itself controllable.

Once the relocation features of the memory management unit are enabled, the conversion from 16-bit virtual addresses to 18-bit physical addresses is accomplished in a

slightly more complicated manner: the high 3 bits of the virtual address are used to select the PAR whose contents shifted left 3 bits are then added to the low order 12 bits of the virtual address to form the 18-bit physical address. [13] If the access is for an instruction fetch or if D-space is disabled, then the I-space PAR is used; otherwise the D-space PAR is used.

When the memory management unit is enabled, there is a strong interaction between the protection it affords and the protection provided by mode-dependent instructions. Four basic machine instructions, MTPI (Move To Previous Instruction space), MFPI (Move From Previous Instruction space), MTPD (Move To Previous Data space), MFPD (Move From Previous Data space), make use of the memory management facilities to read or write specific locations in the virtual space of the program which was last in control of the machine, as specified by the previous mode bits of the PS. This ability is, of course, tempered by the accesses

---

[13]. More precisely, 1) the high 3 bits of the 16-bit virtual address are used to select one of the 8 memory page address registers to be used in forming the physical address; 2) bits 6 - 12 of the 16-bit virtual address are used to select the 32 word block number relative to the beginning of the page frame; 3) this virtual block number is added to the beginning block number of the page which is contained in the selected PAR to yield the physical block number; and, 4) finally, the 12 bits of the physical block number are interpreted as the high order 12 bits of the physical address and the low order 6 bits of the physical address.

allowed by the appropriate PDR's associated with the previous program space (i.e., if the user mode program lacks write access to its virtual page 0, the supervisor program cannot write in user virtual page 0 via MTPD or MTPI instructions). The ability of the supervisor mode program to read and write into the user mode program's virtual space is crucial to efficient virtualization from the viewpoint of the VMM. [14]

Since one of the most obvious security-relevant aspects of almost any operating system is its I/O mechanism, the underlying I/O architecture of the system's basic hardware is of extreme interest and importance. Three aspects of the machine's I/O mechanism are of particular interest: 1) how I/O is initiated, 2) how I/O interacts with the other basic protection mechanisms of the hardware, and 3) how interrupts are handled.

I/O devices on the PDP 11/45 -- indeed, I/O devices on most machines -- operate with absolute, physical memory addresses and are not channeled through the memory mapping

---

[14]. Actually this could be done by appropriately sharing memory segments between the VMM and the VM instead of using MFPI,MFPD,MTPI,MTPD instructions. Such sharing is complicated, however, since the location which needs to be read or written can, in general be arbitrary. Thus kernel intervention is necessary to map supervisor addresses appropriately. The crucial point here is that the VMM often needs to read and modify VM data.

mechanism offered by the memory management hardware. In order to guarantee the integrity of both memory and device segments against accidental or malevolent unauthorized modification by I/O devices operating independently of the protection provided by the memory management unit, it is necessary that the kernel closely control the operation of all I/O devices.

On most machines, control over I/O is rather easily maintained since instructions which are capable of starting, controlling, or interrogating I/O operations are grouped by the hardware into a category of instructions classified as privileged instructions and are unavailable for use by programs operating in non-privileged mode. This is not the case with the PDP 11/45, however: the machine has no explicit start-I/O instructions, as is the case in many other computers. Instead, almost any instruction in the machine is capable of initiating I/O, by reading or writing in special cells (device control registers) located in the uppermost 4K words of the machine's memory. [15] Whether a given PDP 11/45 instruction is I/O sensitive is context-dependent upon whether it references a location in the upper 4K associated with a device.

---

[15]. This is the section of memory which is mapped "specially" when the memory management unit is not enabled. It is accessed through references to locations 160000 - 177776, when memory management is disabled.

The "normal" methods of controlling I/O through use of the restrictions on the program's ability to execute I/O instructions is therefore not available on the 11/45. An alternate mechanism is available, however, which is in some sense equivalent: by restricting access to the upper 4K words of memory, one can restrict access to the device control registers and limit what I/O can be done. This can be accomplished through appropriate use of the memory management unit to prevent user and supervisor mode programs from accessing these device registers. Fortunately, all of these registers are grouped together in the upper 4K and hence removing access to I/O devices is conveniently done.

There are, however, other locations in this last page of physical memory which are not directly concerned with I/O, but these other locations are themselves security-relevant and need to be protected anyway. The prime example of such a location is the PS, the access to which must be controlled for the reasons mentioned previously. The registers associated with the memory management unit and its enabling or disabling likewise reside in this upper page of memory.

Most I/O devices on the 11/45 are capable of being operated in one of two modes: 1) interruption mode, or,

2) non-interruption mode. When a device is started with its "interrupt-enable bit" turned on, it operates in interruption mode and when the I/O completes, it causes an interrupt request to the CPU to be generated. When a device is started with its interrupt-enable bit turned off, it operates in non-interruption mode and no asynchronous notification of device completion occurs when the I/O completes (i.e., no interrupt to the CPU is generated). [16] In both cases, device completion is usually further indicated by the hardware setting of the "done" bit in the status registers of the given device.

Although an interrupt may be generated upon device completion, it is possible that the interrupt may not occur immediately, although interrupts often "preempt" the CPU. This delaying of the interrupt is possible trough appropriate setting of the "processor priority" bits contained in the PS.

---

[16]. This description is a slight simplification of what really occurs. Actually, for most devices, an interrupt will be generated whenever a transition occurs after which both the "device is ready" (or done) bit and the interrupt-enable bit are set to true. "Interruption" and "non-interruption" mode are simply terms used here to convey the logically interesting ways in which the device can be said to operate and are not terms one would necessarily expect to find in any reference manual for PDP 11/45 I/O devices.

Device interrupts are generated at one of four priority levels. [17]

When an interrupt is generated, then, it is generated at a specific priority level associated with the device. If the current processor priority (specified in the PS) is less than the priority of the interrupt, then the interrupt occurs immediately, preempting the CPU after it finishes the instruction (if any) it is currently processing. The current PC and PS after instruction completion is saved in temporary hardware registers, a new PC,PS pair is fetched from the interrupt vector associated with the device causing the interrupt, the saved PC and PS are then automatically pushed onto the stack appropriate to the current mode bits in the new PS [18], and execution continues at the instruction indicated by the new PC.

---

[17]. Although the PDP 11/45 has 8 levels of processor priority (priority 0 - priority 7), only priorities 4-7 are available to I/O devices. The lowest four priority interrupt levels can be asserted only by programmed interrupt requests initiated through the use of the Program Interrupt Register, a special cell in the upper 4K.

[18]. The stack upon which the old PC and PS are saved is the stack which will be in effect after the interrupt is taken. It is the stack which will be used by the routine which has been designated (by placing its PC in the interrupt vector) to handle the interrupt.

If the current processor priority is greater than or equal to the priority of the interrupt, then the interrupt is queued. [19] Interrupts which have lower priority than the processor are inhibited until such time as the processor priority is lowered sufficiently to allow them to occur.

Manipulation of the processor priority can be accomplished through one of four basic mechanisms: 1) explicit manipulation via the SPL instruction; 2) via setting of the processor priority bits in the PS by explicitly modifying the PS; 3) asynchronously via the interrupt or trap mechanism through appropriate selection of the processor priority bits in the new PS portion of the interrupt or trap vector; and, 4) through appropriate selection of the processor priority bits in the word from which the PS is loaded as a result of the execution of an RTI or RTT instruction.

By appropriate setting of the processor priority bits, programs may execute anywhere from the range of arbitrarily

---

[19]. "Queued" is not really the right word for the action which occurs here. "Remembered" is probably more accurate. A number of interrupts may be waiting to occur at the same time, but their ordering is by priority and not by the time of their initial generation. Interrupts at the same priority level are ordered by the "closeness" of their associated device to the processor on the bus.

interruptible (priority 0) up to strictly non-interruptible (priority 7).

The final item of interest with regard to the basic hardware is how it handles exceptional situations and errors, whether they be hardware or software induced. On the 11/45, these exceptional conditions are those which result in the occurrence of a trap -- an event similar in many ways to an interrupt, but one which does not occur asynchronously and which does not operate within the framework of the priority structure used by interrupts. [20]

When the hardware detects the occurrence of a trap, it saves the PC and PS at the time of the trap, loads a new PC and PS from the trap vector associated with the trap which has been detected, stacks the old PC and Ps on the stack for the program mode receiving control, and resumes execution at the instruction indicated by the new PC -- exactly as is the case with interrupts once they are allowed to occur.

---

[20]. There is an explicit ordering among traps and interrupts, however, in the instances in which multiple exceptional events occur simultaneously. Although a program can prevent interrupts from occuring by maintaining its priority at a sufficiently high level, it cannot prevent the occurrence of traps.

Traps on the 11/45 can be grouped into categories: 1) those due to software failures malfunctions, 2) those due to CPU hardware failures malfunctions [21] , and, 3) those specifically requested by the software.

Traps falling into the first category are odd addressing error, bus timeout errors (typically caused by attempts to reference non-existent device registers), memory management violations, stack overflow error, non-existent instruction errors, and reserved instruction errors.

Traps in category two are due to power failure and parity errors, primarily, although CPU malfunctions often manifest themselves through undeserved errors in the first category. [22]

Traps in category three result from the explicit execution of the EMT, BPT, IOT, and TRAP instructions.

---

[21]. Device malfunctions occur as well, of course. These malfunctions, in some instances software programming errors on the device, typically are reflected through device interrupts, however, and not through traps.

[22]. Often a malfunctioning CPU will cause bus time-outs, reserved instruction traps, etc. when there is no real reason for those traps to be generated.

2.2 Hardware Modifications to the UCLA PDP 11/45

The UCLA virtual machine system was built to run on a Digital Equipment Corporation (DEC) PDP 11/45 computer. The PDP 11/45 computer, however, as originally marketed by DEC is not virtualizable since certain sensitive instructions do not trap when executed in user or supervisor mode [23] The UCLA PDP 11/45 was therefore modified in order to make it virtualizable. [24]

The major modification consisted of additional logic to make certain sensitive instructions trap when executed in user or supervisor mode. [25]

In addition a user stack limit register was added [26] and provision made to allow traps to be sent directly to the supervisor mode program when caused by a user mode program rather than requiring that these traps be fielded

---

[23]. The reader is referred to [POPEK74B] for a definition of what constitutes a sensitive instruction, as well as the formal requirements for virtualization.

[24]. A complete description of these modifications can be found in [DEC74].

[25]. In the unmodified CPU, certain instructions which are sensitive become no-ops when attempted in user or supervisor mode rather than causing an abort or trap.

directly by the kernel. This was accomplished by the addition of a set of alternate trap vectors which can be enabled when the virtual machine modification is enabled and which are interpreted with the memory management unit enabled (i.e. use the memory mapping mechanism). [27] However, the occurrence of certain traps are highly indicative of hardware malfunctions. Therefore, even with the modified hardware, the alternate interrupt vectors are set so that some traps still go directly to the kernel, regardless of the processor mode at the time of the trap. Examples of these sort of traps include memory parity traps and power fail traps.

All interrupts, however, whether they occur while the processor is operating in kernel, supervisor, or user mode must go to the kernel. This is necessary not only because

---

[26]. The user stack limit register is not security relevant, except for the fact that it was placed in a most inconvenient spot as far as the kernel was concerned, and hence will not be described here. The user stack limit is, however, necessary for virtualization of the kernel stack limit register of the unmodified machine (and the modified one as well).

[27]. The ability to send user mode traps directly to the supervisor portion is not strictly necessary. It is more of an efficiency issue than anything else since one can channel all traps and interrupts into the kernel and then have the kernel reflect them back to their proper destinations. This would, of course, require additional kernel code.

the kernel must for security reasons control all I/O, but also because the interrupt is not necessarily intended for the currently running process. [28]

---

[28]. The modified hardware has access only to the relocation registers of the currently running process. Since the hardware interrupt mechanism operates within the framework of memory management, any interrupt which is allowed to pass directly to the supervisor portion would end up in the supervisor portion of the currently running process, which is not necessarily the process which requested the interrupt. Although the kernel has full knowledge of which process the interrupt belongs to, the hardware itself does not.

2.3 Security Objects

Once the decision has been made to provide protection, one is left with the task of defining the limits of the protection to be offered: one must define the units of protection to be offered, as well as the various levels of protection which will be supported. Before doing so, it is convenient to classify portions of the system into objects. If all system entities can be so classified and if the system of classification is sufficiently flexible to encompass a large number of semantically dissimilar entities in a homogeneous manner, then by choosing these 'objects' as one's units of protection, one is able to provide a homogeneous protection foundation for the system.

The question then is, what are the basic objects in the system? What is the smallest protectable unit? There is a natural trade-off here between providing protection in extremely small units, which might be desirable from a flexibility standpoint, and providing protection in large units, which may be desirable from an implementational and efficiency viewpoint-- that is, there is a trade-off between user flexibility and reasonable implementation.

One could, for instance, choose a single word to be the basic protectable unit. One then could succinctly classify the system users according to whether or not they

are able to read word X. One then has complete control over which system users can access particular words of one's data.

There are some problems with providing protection on a single word basis, however. Although the notion is logically feasible, it has some serious implementation impacts: 1) it implies a tremendous disk storage overhead for representing the protection policy, since it requires the existence of an access list for every protectable word in the system [29] ; and 2) it implies tremendous kernel overhead if the system's hardware base is not flexible enough to provide convenient single word protection [30]

One must tread a thin line then between protection flexibility and implementability in selecting one's unit of

---

[29]. Such overhead is clearly intolerable if one wishes to protect these access lists in a manner identical to that afforded all other objects in the system since one now needs an access list to protect the access list to protect the access list, ad infinitum. This is clearly unimplementable in any system with a finite amount of storage since each word of storage requires at least one additional word to protect it, which in turn must itself be protected, etc. Such a scheme leads to infinite recursion. The problem with protecting single, individual words in this manner is that the amount of information needed for flexible control of the protection is unlikely to fit within a single word; hence the protection for a word must be spread throughout a group of other words which must be likewise protected. By choosing one's smallest protectable unit to be large enough to contain sufficient protection information within a single unit, the recursiveness of this problem can be effectively limited.

protection: it must be small enough to be a logically protectable entity, yet it must likewise be large enough for its associated protection policy to be reasonably represented and its protection enforcement to be implemented in a reasonably efficient manner. [31]

But before one can choose a physical format for an object, it is first useful to examine the sort of logical properties one would wish to attach to an object. What sort of protection is desired? One certainly wishes to limit and control access to collections of data, both in memory as well as on secondary storage. One might wish to control access to physical resources such as memory, tape drives, disks, etc. One might wish to limit the influence one user might have on another, such as restricting their ability to communicate with each other.

What sort of objects do these desires imply? Certainly one might consider collections of data in memory and on secondary storage as objects. Additionally one would

---

[30]. The overhead is probably enormous even if the hardware makes single word protection fairly convenient. The PDP 11/45 hardware doesn't even come close to the "fairly convenient" level.

[31]. This implies, of course, that the choice of one's smallest protectable unit is likely to be extremely dependent upon the underlying protection facilities of the system's base hardware.

probably classify devices as objects, and finally, users seem to be another sort of object. If these three types of objects are implemented, then the resulting protection can be summarized as follows: 1) Data collections are protected from devices and users; 2) Devices are protected from devices and users; and 3) Users are protected from devices and users.

These three objects can further be divided into two classes: 1) passive objects, and 2) active objects. Data collections are, of course, passive objects, by nature, while both devices and users are capable of being alternately active or passive, depending upon whether they are being acted upon or performing the action directly. Clearly these three types of objects are not homogeneous. However, if they do represent all the possible types of objects in the system then little has been lost by subdividing one's protection mechanism into three categories.

Objects in the UCLA Virtual Machine System can be classified into four categories: 1) collections of code/data [32], called segments, 2) devices, 3) processes, and 4) the kernel.

---

[32]. The distinction between code and data on the PDP 11/45 is strictly one of semantics. If one attempts to execute it, then it is code. If one explicitly tries to read or write it, it is data.

2.4 Processes

A process in the UCLA Virtual Machine system is a single object and consists of a user/supervisor pair, where the user portion is optional. All protection is maintained on a per process basis. No protection is guaranteed between the user and supervisor portions of the process pair, although some protection is afforded the supervisor portion from the user portion automatically through the mode protection and relocation features provided by the basic hardware. [33]

Each process has at its disposal 6 general purpose registers, two stack pointer registers, one program counter, one program status word, 32 relocation registers, and some number of 4K-word blocks of memory. A process is allowed to execute in either user or supervisor mode. Kernel mode execution is reserved exclusively for the security kernel. The supervisor mode portion of the process is the only portion of the process which is allowed to communicate directly with the security kernel. The

---

[33]. This is of considerable importance if one wishes to share supervisor code portions between multiple processes in insuring that the actions of a malicious user portion of one process is unable to adversely affect the operation of another process sharing the same supervisor portion.

knowledgeable concerning the details of the kernel/process interface. [34] Figure 2 illustrates the decomposition of the active system components according to execution mode.

At any given instant in time, only one member of the user/supervisor process pair is in control of the process. Which member of the pair is in current control of the process can be determined by examination of the program status word associated with the process. [35] If the current mode bits of the program status word indicate supervisor mode, then the supervisor portion is in control. If the current mode bits specify user mode, then the user portion is in control.

Where the program status word is located at any given moment in time depends upon whether the given process is running. If the process is not currently running, then the program status word, along with the contents of all relocation registers, stack pointers, program counter, and general registers, is stored in the kernel's process table

---

[34]. The kernel/process interface will be described in the following chapter.

[35]. This program status word should not be confused with the processor status word (PS), although in some instances the two are identical. The processor status word is a hardware register and is used by the hardware when performing mode-dependent operations; the program status word is used by the kernel in performing process mode-dependent checks.

entry for the given process.

If the process is currently running, then the program status word is found in one of two places, depending upon whether the kernel or the process itself is in current control of the actual CPU. If the current mode bits of the real hardware processor status word indicate that the CPU is currently executing in kernel mode, then the program status word and the program counter for the most currently running process [36] reside in the kernel's process table for the given process.

If the current processor mode given by the hardware processor status word is not kernel mode, then the program status word for the process is resident in the CPU processor status word (PS) and the program counter for the process is found in the hardware program counter (R7). In either case, the general registers, relocation registers, and stack pointers of the currently running process reside in the appropriate hardware locations. [37]

Once multiple processes enter into the system picture, there arises the necessity of addressing the issue of multiplexing the CPU between the various processes running

---

[36]. Since the kernel is currently running and is not considered part of a process, no process is actually running. The most currently running process is the process which was running just before the kernel usurped control of the CPU.

under the kernel. CPU scheduling, however, tends to be something that generally undergoes an extensive amount of fine-tuning as machine usage evolves and priorities and constraints change. Scheduling algorithms also involve a fair degree of complexity. Most importantly from our viewpoint, however, scheduling decisions are related to security only in the sense that they may result in denial of service at the worst, and, as such, can be classified as viability issues. Hence, while it is conceptually possible to prove scheduling code correct, there seems little real gain in placing such code within the kernel itself.

Indeed, there are several good reasons why such a step should not be taken: 1) As previously mentioned, scheduling algorithms rank among the most modified portions of code in existing operating systems and changing the scheduling policy might necessitate the reverification of large portions of the kernel; 2) scheduling questions are not data security questions and hence do not belong in the kernel if they can be made elsewhere; and, 3) if later it

---

[37]. This is possible since the 11/85 has two independent sets of general registers, 3 stack pointers, and 3 sets of relocation registers. The kernel utilizes the kernel stack pointer and relocation registers and register set 0, while the process uses the user and supervisor stack pointers and relocation registers and register set 1. The hardware general registers and relocation registers dedicated to the process are saved and restored only when processes are switched. The PC and PS, however, must be saved whenever the kernel is running (since they are shared between all program modes) and are found in the kernel's process table entry for the given process.

becomes desirable to verify the scheduling code, isolating it outside of the kernel in a module concerned only with CPU scheduling should simplify the proof task.

For these reasons, the decision was made to remove CPU scheduling code from the kernel. Instead, the scheduling function is delegated to a special CPU scheduling process running under the kernel, which is periodically invoked by the kernel in order to perform its scheduling task. Actual saving and restoring of process context (program counter, program status, relocation registers, general registers, and stack pointers) must of course be performed by the kernel in order to guarantee separation of process data.

Processes can be divided into three categories: 1) privileged processes, 2) system processes, and 3) non-privileged processes.

Privileged processes differ from system processes mainly in that they are allowed to communicate certain information to the kernel that will subsequently be used by the kernel in security relevant ways. These processes are those which must eventually be proven before the security of the entire system is guaranteed. Examples of privileged processes are the initiator and the updater.

System processes are those which possess the ability to perform actions which affect the performance of the

system, but which possess no other special security relevant privileges. Examples of system processes include the CPU scheduler and the shared device schedulers.

Non-privileged processes are the normal, everyday type processes. They include the Virtual Machine Monitor/Virtual Machine (VMM/VM) process pair.

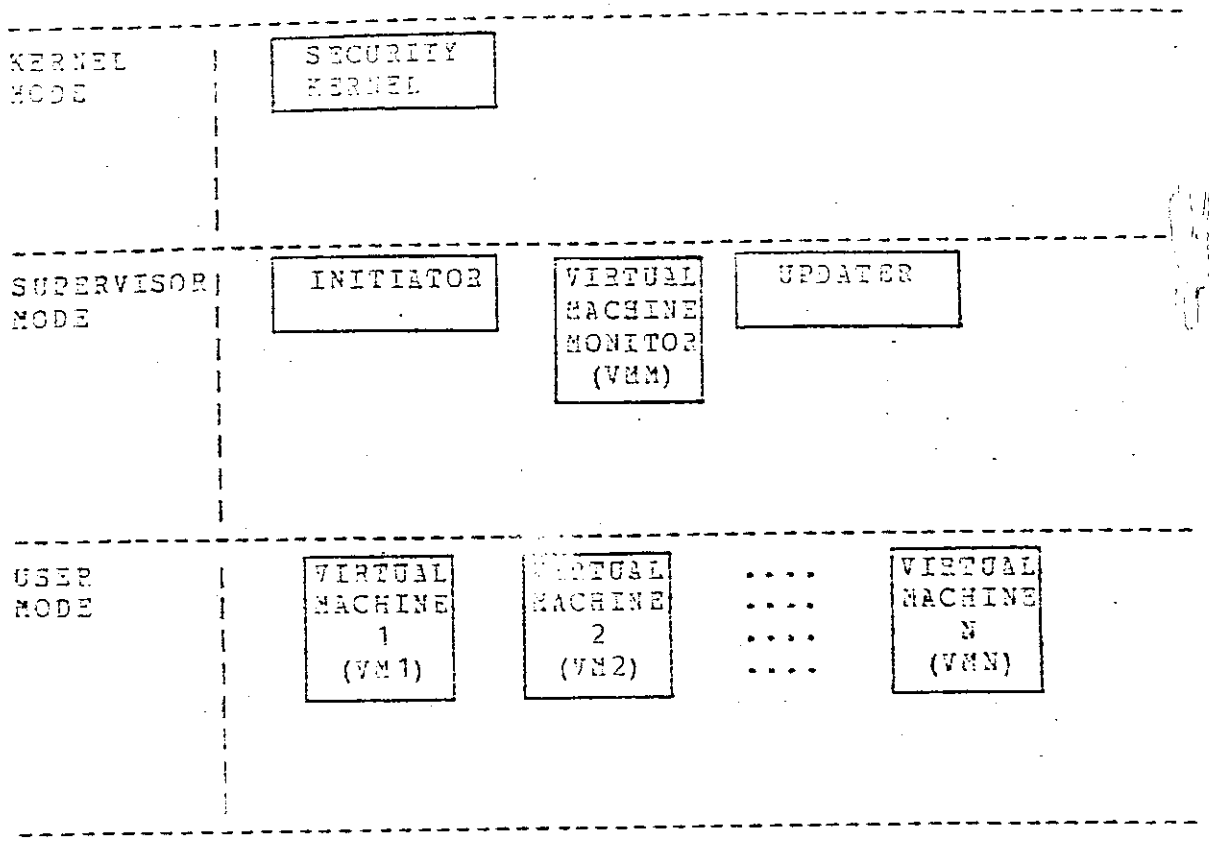

Figure 3: System Decomposition By Execution Mode

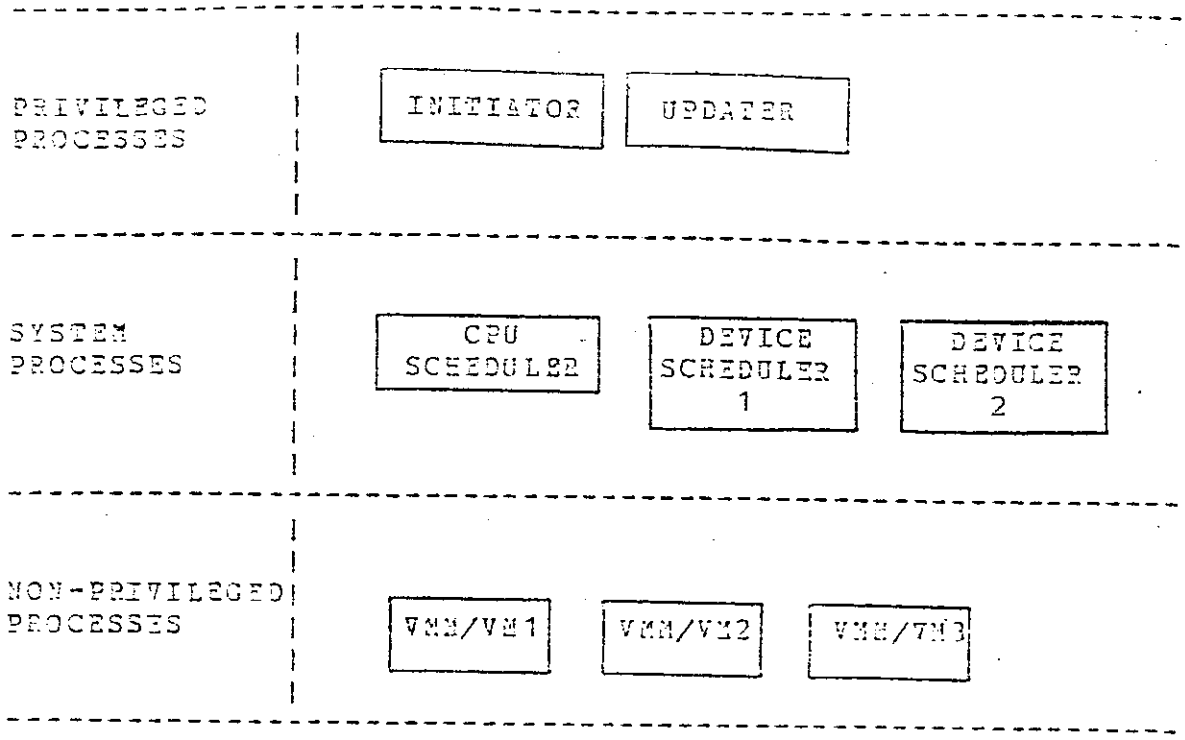

Figure 4: System Decomposition By Process Privilege

2.5 Devices

Clearly, in order to construct a robust system, it is necessary to partition certain real devices (disks, for example) into some number of smaller units, then allocate multiples of these smaller units to individual processes, rather than dedicating the entire device for the exclusive use of a single process. In allowing for such sharing of the device, it was necessary to devise some means of coping with the device contention problem which results from multiple users attempting to use a single device simultaneously. Again, this was a scheduling problem and hence not one that belonged in the kernel. Instead, it was pushed out of the kernel and onto the shoulders of special processes called shared device schedulers.

In order to provide a solution to this contention problem, device schedulers for those devices which were to be partitioned were introduced into the system. This might seem a rather trivial, uninteresting decision on the surface, but it was one which had rather severe and important consequences relative to the overall system design. Had no attempt been made to provide for shared devices, several kernel calls would have been unnecessary and a number of the more difficult design problems eliminated. On the other hand, one suspects that the

resulting system would have been a much less useful one. [38]

Since the ability to restrict utilization of certain devices seemed a likely candidate for a security policy decision, it is necessary that the final say-so in device allocation be given by the kernel itself. How to schedule a given non-dedicated device between competing users, however, falls again within the productivity-viability area and has no relation to data security, provided that the kernel does indeed maintain the security of any sub-divisions of the device storage area between processes.

As was the case with CPU scheduling, device scheduling is often a candidate for fine-tuning and honing operations, and the same modification-reverification arguments that can be applied to CPU scheduling are equally valid when applied to device schedulers. Hence device schedulers join the CPU scheduler as additional processes running outside of the kernel. Thus, while the majority of I/O devices in the system are dedicated, non-shared devices, a number of devices (notably the disk) are capable of being shared between multiple processes.

---

[38]. Not everyone can afford to dedicate an entire disk to a single virtual machine.

Devices in the UCLA Virtual Machine System can be divided into two logical categories: 1) dedicated devices, and 2) shared devices. [39] A dedicated device is a device which may be allocated to an arbitrary process and which can not be used by any other process while it is so allocated. A shared device, on the other hand, is allocated to a specific process which may then share the device with other processes, but which retains control over the initiation of any I/O connected with the device. [40] Furter discussion of how I/O to shared devices is accomplished ca be found in the sections dealing with the Request-I/O and Start-Indirect-I/O kernel calls.

A device becomes allocated to a given process upon the successful completion of an Attach-I/O-Device kernel call and remains allocated to the given process until it is released (via a Release-I/O-Device kernel call) or until the process requests to attach it to another process. A process which has attached an I/O device may be allowed to

---

[39]. Logically, devices can be separated into dedicated and non-dedicated devices. Physically, however, all devices are 'dedicated' devices, in the sense that a device has only one owner or user at any given moment. Strictly speaking, non-dedicated devices are really dedicated devices which the system policy "pre-allocates" to specific processes.

[40]. This 'specific' process is the device scheduler for the given shared device.

initiate I/O to or from the device by executing a Start-I/O kernel call, or receive status information on the device by executing a Status-I/O kernel call.

A process which is allowed to use a non-dedicated device requests initiation of I/O to the non-dedicated device by executing a Request-I/O kernel call. This causes the I/O request to be transmitted to the owner of the non-dedicated device, which may eventually cause the I/O to be initiated by executing a Start-Indirect-I/O kernel call. Error or completion information is returned to the initial requestor upon device initiation and completion, respectively.

2.6 Segments

A segment in the UCLA Virtual Machine System is an arbitrary contiguous collection of 4K words which resides either in memory or on secondary storage. [41] Five major access capabilities can be associated with segments: 1) no access, 2) read only access, 3) read/write access, 4) read/execute access, and 5) read/write/execute access. Protection is maintained homogeneously on a per segment basis. All words contained in a single segment are protected identically-- that is, if word X1 and word X2 both reside in segment X, then any user having access Y to segment X will possess access Y to both X1 and X2.

The kernel supports five major operations on segments: they can be 1) created, 2) destroyed, 3) attached, 4) swapped, and, 5) released. Once a segment has been created, it continues to exist in some form, either in

---

[41]. This size is in some sense arbitrary, but the 4K figure is particularly convenient in that 4K words is the largest size segment that can be protected by a single relocation register using the 11/45 memory management hardware. Additionally, the choice of 4K word segments allows the virtual machine monitor to simulate a physically contiguous virtual address space to the VM. If a segment size of less than 4K words is chosen, the resultant address space will contain "holes" in it. The reader is referred to the PDP 11/45 Processor Handbook [DEC73B] for further details.

memory or on secondary storage, until it is explicitly destroyed. Existing segments may be accessed directly by processes once they have been attached and brought into memory and remain directly accessible until they are swapped out of memory or are released. Existing segments which are not core-resident can be indirectly accessed via I/O devices associated with the process under certain circumstances.

The kernel was designed to provide controlled, limited sharing of data between processes. Under "normal" circumstances, the resources available to a given process at a give time are disjoint from the resources accessible to other processes at the same moment. However, minor modifications in the security policy make limited data sharing between mutually willing processes possible through the use of "shared" segments.

A shared segment is conceptually no different from a non-shared segment. The only distinction between the two types of segments is that the protection data associated with the shared segment indicates that it may be accessed by more than one user.

It is believed that the shared-segment mechanism provides enough of the basic communications tools to enable

the construction of some message handling system above the security kernel. [42] There is the necessity, however, of providing some way of initially establishing a communications link. Since the kernel maintains process isolation except for shared segments themselves, it is necessary to place this initial communications channel somewhere in the kernel. Thus some special means had to be designed to allow processes to communicate among themselves. The extent of such kernel communication support did not need to be large, however, and a simple-minded Send-Message kernel call was provided to meet those needs.

---

[42]. A bare minimal set of communications primitives will be furnished by the kernel: sleep, wakeup, and send-message.

2.7 Complete System Structure Summary

The general structure of the system is fixed. At the lowest, most basic level of the system, running upon the bare hardware, lies the security kernel, providing inter-process protection, communication, and security enforcement. At the next outermost level, outside of the security kernel, lie the other basic components of the system: the virtual machine monitor, to simulate to the virtual machine those capabilities of the real machine usurped by the kernel, the initiator to identify users to the kernel and create, start, stop, and destroy processes, the CPU scheduler to multiplex the CPU among processes running outside of the kernel and make decisions as to how memory management should be handled, device schedulers to multiplex shared devices among competing processes, and the updater to maintain and control the protection data used by the kernel in making security decisions. This structure is illustrated in figure 5.

"All" that remains is to thoroughly examine the individual components, isolate the security enforcement issues involved in each, and insure that the kernel contains some means of providing them.

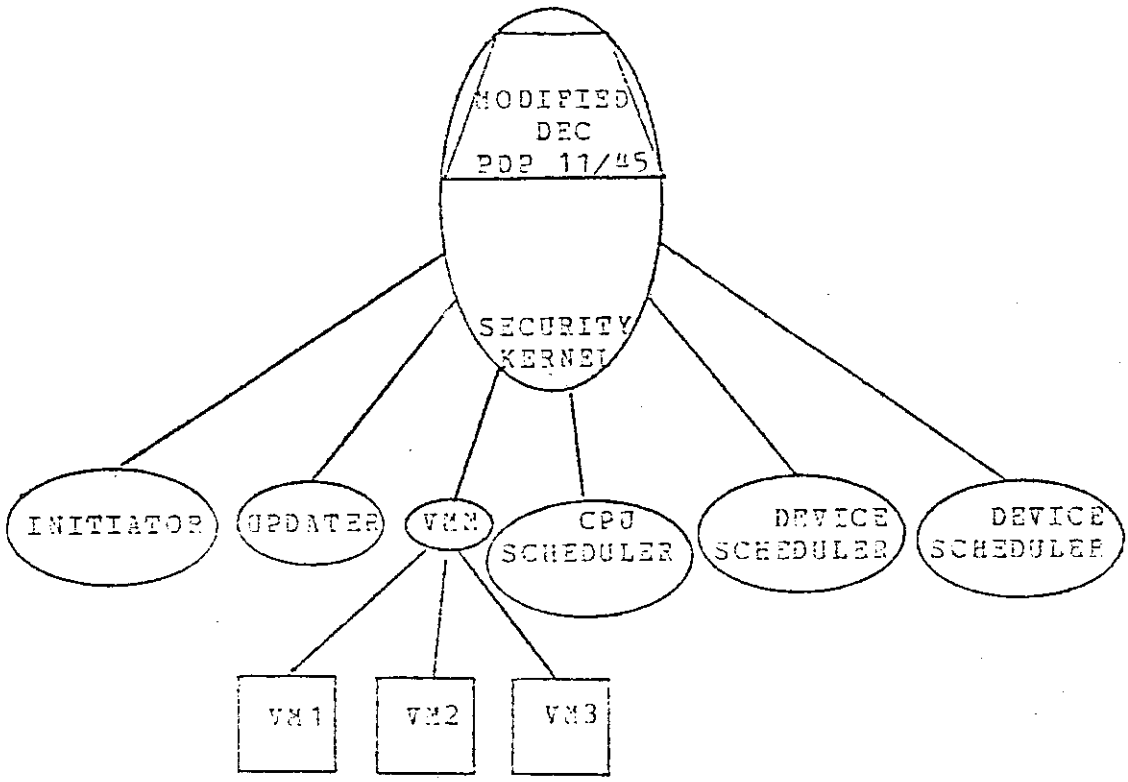

Figure 5: UCLA VM System Structure

Chapter 3: Basic Kernel Design Decisions

3.1 Memory Management

Quite early in the design process the question arose regarding what place memory management should have in the security kernel. From the first we were hesitant to place the responsibility for providing virtual memory support at the lowest level in the kernel. Not only does memory management tend toward complexity, but it often has the scheduling flavor about it if one is to handle such things as making page faults and the like invisible by doing demand paging. It just did not have the feel of the sort of code which needed to be a *necessary* part of the kernel.

The real drawback of placing the lowest level of memory management support in the kernel was that it raised the interruptibility issue in all of its gory details: if one were to make full use of the memory management facilities now placed in the kernel, it seems only reasonable to allow the kernel itself to rely upon them. Yet now one finds oneself in the awkward situation where certain (and now almost assuredly, arbitrary) portions of the kernel need to be interruptible in order to function properly.

So what if the kernel now needs to be interruptible at

arbitrary points? Why is that such a big drawback? It might not be such a problem if the state-of-the-art of program verification techniques were more advanced, or if it were significantly easier to write code in such a fashion to make it arbitrarily interruptible. Unfortunately, interrupts are capable of instigating side-effects of the most confusing and intricate sort, and side-effects in even their simplest and most rudimentary forms are extremely ill dealt with by current program verification techniques.

Besides, was memory management really a necessity for the kernel? Not if the original size expectations about the kernel could be realized. Why would a small kernel need virtual memory support for itself? Its code should clearly fit in the available space. But what about its data? Was there any reason to believe that the kernel's necessary data segments could not always be kept core-resident? Unfortunately the answer seemed to be yes if one wishes to handle security policy issues in any reasonable manner on a relatively large system. But this represents a relatively specialized problem and did not seem sufficient motivation to sacrifice kernel uninterruptibility if a simpler, perhaps more restrictive, solution could be discovered.

Indeed, an alternative solution, one which pushed the responsibility for bringing protection data segments into core onto processes running outside of the kernel, was

devised and will be discussed at length in the section on capability faulting. Additional discussion of the notion of capability faulting can be found in [POPEK74C, POPEK75].

Yet while it seemed most reasonable to believe that the kernel could function properly without benefit of underlying memory management support, it seemed not nearly so reasonable to believe that the rest of the processes in the system could manage without some sort of help in that area from the kernel. 124K, after all, is not much memory to share between a CPU scheduler, initiator, updater, disk scheduler, and several processes other than the system components. [1] And, of course, one has to include the kernel's code in that 124K figure.

It seemed reasonable to expect that some sort of swapping facility would eventually be necessary, and although the initial version of the system might not utilize it, enough thought had to be given to swapping considerations to demonstrate that the entire mechanism was feasible.

Having thus perceived the necessity of placing at least some minimal memory management support within the kernel, it next became a question of determining the

---

[1]. There is little point in bothering with verified security if one is merely protecting a single process from itself.

magnitude of such support. Following the "keep it small" design criterion, we were determined to include as little extra kernel code as we could manage to get by with and yet still meet viability constraints. [2] What then was the least we could get away with?

It was almost immediately clear that the decision as to the appropriate time to swap a given segment was one which the kernel did not wish to make. [3] Equally as obvious was the fact that which segment to swap is not a decision which should be left up to the kernel either. [4] Neither what to swap, nor when to swap it are really security issues at all, as long as the kernel enforces some minimal constraints about what can not be swapped when.

Well then, if the kernel is not concerned with why a segment is being swapped, what segment is being swapped, or when a segment is being swapped, then what is the kernel

---

[2]. We are not so much concerned with making any mechanism easy to implement, simply with making it possible to implement.

[3]. All of the fine-tuning arguments apply in full force. Policy that may be expected to change does not belong coded into the kernel.

[4]. How in the world should the poor tiny kernel know what segment will be needed next? If it has been busy keeping track of such things, then it is probably a lot larger than one can expect to get away with proving.

concerned with? The answer is that the kernel is concerned with how the segment is being swapped, more specifically in from what and into what the segment is being moved. It is interested mainly in assuring that segments are labelled correctly.

Although the majority of the problems of swapping are not the concern of the kernel, the swapping mechanism does indeed involve I/O and has severe security implications. It is of prime importance that the kernel be totally correct in its association of in-core segments with process names. If the kernel can be led to believe that in-core segment X belongs to process Y, when in reality it belongs to process Z, a security violation has occurred. While it is of little consequence to the kernel when or for whom swapping occurs, it is of vital importance that the kernel correctly identify the segments currently residing in memory, in order that protection checks are made upon the proper basis.

A slightly different mechanism is therefore necessary to handle swapping than the normal I/O mechanism. There is a logical difference in the actions required of the kernel when a disk request is a swap request than when a disk request is a normal disk request: A normal disk request does not change the name of its target core or disk segment; a swap disk request does. Since the names of core

and disk segments are security relevant and controlled by the kernel, the kernel must take additional actions both upon the initiation of the request and upon its completion.

Failure upon the kernel's part to take suitable precautions at the beginning of a swap out request, for instance, is readily seen as a possible security flaw. Consider the following scenario: core segment X was requested to be swapped out. If the kernel has not taken appropriate notice of this action, it is possible for the resulting disk copy of the segment to become corrupted as a result of intermediary I/O done into the core segment between the time the swap request was initiated and the time it eventually completed. One can imagine a malevolent CPU scheduler behind such a move, one which has decided to swap out some portion of a process which it has decided to run nonetheless.

If the kernel has not acted appropriately to make that segment unreachable to all processes during the time of the swap, even worse consequences can arise during the eventual swap-in, resulting in data belonging to the former owner of the in-core segment being transmitted to the new owner of the segment.

Interestingly enough, from a pure data security standpoint and ignoring viability issues, it is not really

necessary that the kernel swap correctly -- that is, swap the given core segment to or from the indicated disk segment. All that is really necessary is that no expansion of the capabilities of any user is accomplished as a result of the swap. No security violation results if segment X belonging to user A is swapped into the disk space reserved for user A's segment Y so long as that swap has not given any user access to X which it did not previously possess. If B previously had no access to X, but was allowed to share Y, then this would be a violation.

Such violations are fairly subtle, and it is probably much simpler to show that the kernel swaps correctly rather than showing that any incorrect swaps do not violate security constraints. But even here one merely need show that memory frame X is transfered into disk frame X and not disk frame Y. It does not require that the resultant contents of disk frame X appear identical to the previous contents of memory frame X. The kernel might have inverted X's contents in the process of the swap but such an inversion would not be considered a data security flaw. [5]

The major responsibility for memory management in the

---

[5]. It is extremely unlikely that the kernel would invert a segment during a swap, however; inverting it is much more complicated than merely copying it.

UCLA Virtual Machine system, then, is placed on the shoulders of the CPU scheduler. Process requests to have segments swapped in and out are communicated to the CPU scheduler via the Send-Message kernel call and the actual swapping eventually requested by the CPU scheduler.

3.2 Capability Faulting

Although we are primarily concerned with *security enforcement* and not *security policy* in the design of our system, certain minimal viability constraints necessitated the examination of some facets of policy questions during the design process. Since it is necessary to protect some large number of disk segments spread through some large number of possible users, a totally in-core protection data base is an unreasonable choice -- certainly one which does not readily lend itself to system resource expansion.

On a system with a large number of available disk segments, it is conceivable that a significant amount of memory would be wasted merely in keeping the protection data core-resident, thus significantly reducing the memory resources available to the remainder of the system, perhaps to zero. Such a constraint seemed unreasonable, so some viable alternative was sought.

It is obvious that some portion of the protection data should be core-resident for efficiency's sake. A scheme that would require swapping of protection data from disk to memory each time a process asked to do anything is clearly intolerable from an efficiency viewpoint. However, it was just as clear that not all of the protection data could be kept in memory all the time. A compromise between the two

extremes seemed to be in order.

At this point another crucial observation can be made: although it seems more convenient to provide protection on a per object basis, most protection questions seem naturally stated on a user-oriented basis. While it is most convenient to store the data on an object basis, only a small portion of the entire set of protection data would be of interest at any one time during the normal operation of the system-- not all users will be actively accessing all objects at any given moment.

If one views the protection data as being represented in a Lampson access matrix with passive objects labelling the columns and active objects labelling the rows, and then attempts to collapse the matrix into triples to be stored either by their row or column label, one discovers that while the data is most properly stored by column, it is most conveniently accessed by row. The protection data most usefully kept in core is that data pertinent to the currently active system users.

Next the question arises of how portions of the protection data are brought into core -- at whose request and in what manner. Several alternatives came immediately to mind: 1) The kernel could swap in appropriate portions of the process's protection data at the time of process

creation; 2) the initiator could ask for it to be swapped in at process creation time; 3) the CPU scheduler could ask for it to be swapped in whenever it runs the process; or 4) the process itself could ask for it to be swapped in.

Alternatives one, two, and three potentially involve significant process start-up delay. Alternative two might cause additional complications to the initiator which must eventually be proven correct. Alternative one would necessitate direct I/O operation by the kernel, with resulting confusion due to disk scheduler interference and communication, and reintroduces the kernel interrupt problem previously set at rest by making the kernel uninterruptible.

The fourth alternative, however, possessed none of these drawbacks, although it complicates the user process somewhat and requires that the user process have at least some minimal knowledge of kernel protection data structure, especially the naming conventions for kernel protection data disk segments. It also requires a kernel call to allow the user process to cause a disk segment to be read in on its behalf without having any real access to the segment itself. [6]

The last alternative seemed the most reasonable of the four in spite of the necessity for another kernel call.

The complication to user processes should be minimal, yet efficiency is now lost only when necessary. Under the fourth alternative, a mechanism which we term capability faulting, processes operate as if all protection data pertinent to them is core-resident until they receive a capability fault -- a special kernel call response which indicates that the kernel has insufficient protection data in core to make any decision regarding the legality of the process's current request.

Upon receiving the capability fault, the user process then determines which disk segment contains the required protection information, and sends a message to the CPU scheduler asking for the protection data segment to be swapped into memory. The CPU scheduler then executes a Swap-In kernel call, asking the disk scheduler to read the segment into core.

If the process has requested the proper protection segment, then the next time it attempts whatever action resulted in the capability fault after the segment has been read in, it will receive a more definitive answer from the kernel.

---

[6]. This is done via the SEND-MESSAGE kernel call. The process asks the CPU scheduler to swap the segment in. Originally there was a specific kernel call to do this, the "Request-Segment" call, but was generalized into the SEND-MESSAGE kernel call.

3.3 The Kernel/Process Interface

Communication between the kernel and a process running under the kernel is established via a core resident data area shared between the kernel and the process called the shared kernel/process communications buffer. This shared area must remain in real memory at all times and cannot easily be swapped out for it is often the case that the kernel must send information to the process (interrupt notification, for instance) during periods of time when it is not running (and thus its memory segments not ordinarily guaranteed to be core-resident). This prohibits any swapping of the shared kernel/process communications buffer on a per process basis along with the process.

Since much of the memory management support is layered above the kernel in our system and not beneath it (as is the case in many other systems) and since one of the assertions we would like to be able to prove about the kernel is that traps never occur within the kernel (unless, of course, the hardware is physically failing), it is necessary that the kernel never get a memory management fault. Hence the shared kernel/process communications buffer must be locked into core at all times, though it is not necessary that the shared communications buffer be of the same size as other segments in the system. [7]

The shared kernel/process communications buffer is divided into three logically distinct areas: 1) a read-only portion, 2) a process read/writeable portion, and, 3) a read/write/executable portion.

The read-only portion of the communications buffer is reserved for storing information which may be read by the process, but which for security reasons may not be modified by the process. This area is principally used for storing locked boxes, collections of information which have been bundled up by the kernel and put away for safe keeping until whatever further processing they require has been completed. The most prominent use of locked boxes in the system is in transmitting a process's shared device I/O request to the appropriate device scheduler so that the scheduler can decide when to perform the request without being able to alter the request.

The read/write portion of the communications buffer contains three major sections. The first two sections are concerned with synchronized communication between the kernel and the process. The first of these sections is used for passing kernel call arguments from the process to the kernel. The second section is reserved for passing

---

[7]. Indeed, if this were a requirement, it would seriously restrict the number of processes which could be run in the system.

kernel call responses back from the kernel to the process.

The third section of the read/write portion of the communications buffer contains a first-in-first-out wraparound queue where miscellaneous asynchronous information can be passed from the kernel to the process and inspected at the process's leisure. Responsibility for draining this queue area rests with the process. If queue entries are not processed at a sufficiently high rate, the kernel will discard entries rather than bother with the problem of queue overflow. The problem of insuring that the queue does not overflow is merely a matter of keeping track of the current number of outstanding I/O requests and pending kernel requests and not requesting more than can be safely fit into the queue at any one time.

Although the kernel does not explicitly concern itself with problems of queue overflow, a great deal of careful consideration was given to deciding what sort of information would be placed in the queue, in hopes of providing some assurance that a reasonably "correct" and responsive process would not need to worry about its queue overflowing.

A reasonable question which might be asked at this point is: why should the kernel concern itself with problems of process queue overflow? What bearing does

process queue overflow have on the issues of data security? One is tempted to reply, very little. Such an answer, however, leads right back to the first question: why bother?

The process queue overflow problem would seem to be a question of the "proper" functioning of user processes, something we have never even pretended to guarantee. However, the security properties of the system as a whole as currently designed depend to some extent upon the correctness of certain processes running outside of the kernel, namely the initiator and updater. Thus assurance is required that processes <u>can</u> be written (in particular, that an initiator and updater can be written) which can be shown to function properly. [8]

If there is no way of guarding against queue overflow, then an effective demonstration of the proper functioning of any process which ever needs to use the queue would seem impossible. Since both the initiator and updater must communicate with terminals, hence performing I/O, one suspects that the queue and its contents will be of great

---

[8]. However, even an initiator or updater malfunction caused by queue overflow cannot cause a data security violation to occur, as far as the kernel is concerned. At the worst, information which would otherwise be communicated to the initiator or updater will be lost and some action which was initiated might possibly never be completed.

importance to them. Thus, the queue overflow problem must be addressed to some extent and successfully resolved.

The read/write/execute portion of the communications buffer is used to implement the kernel pseudo-interrupt mechanism and will be discussed further in the section on interrupts.

3.4 Kernel Structure

The kernel can be divided into three parts, based upon the means by which it receives control: 1) the kernel call handler, 2) the kernel interrupt handler, and 3) the kernel trap handler.

These parts are not strictly independent since they share portions of code and data. However, they do operate independently of one another. It is not so much a question of where the current kernel program counter points so much as how it got there -- which of the three entry sequences into the kernel was used.

Entry sequences one and three share the property that when entered through either, the kernel performs some action on behalf of the currently running process (unless in case 3 the trap was a kernel trap). Entry into the kernel through sequence two may cause action on the behalf of any process, or, as is the case of a clock interrupt, no process.

Regardless of the original entry sequence, the kernel will proceed with a given action until its completion. The kernel runs in a totally uninterruptible mode. [9] No other action will be started until the initial action is completed.

3.5 The Kernel Trap Handler

On the unmodified PDP 11/45, the first 1000 octal bytes of the kernel's virtual memory are reserved for trap and interrupt vectors. When traps are generated, the processor ceases execution at its current site of operation and resumes operation at the instruction pointed to by the "new PC" portion of the trap vector in the instruction space appropriate to the "new PS" portion of the trap vector, after first pushing the PC and PS which existed at the time of the trap onto the stack corresponding to the new processor mode. The location of the trap vectors is determined in kernel data space (ie. the new PC,PS are fetched using the memory management registers corresponding to kernel data space locations 0 - 1000 octal). The new processor mode is determined by the new PS mode of the trap vector contents. Virtually no restrictions are made by the hardware on the contents of these trap vectors, although "incorrect" contents may result in the generation of additional traps.

---

[9]. This is accomplished by always entering the kernel with hardware processor priority 7, which means that no interrupts will be allowed to occur until the processor priority is deliberately lowered. This is quite independent of whether the kernel operates with device interrupts enabled, since no device interrupt will be allowed to occur (will ever be seen by the CPU) as long as the processor priority remains at priority 7. The interrupt may be waiting to occur, but will not be allowed to occur until the kernel changes its priority.

The trapping mechanism under the modified CPU is identical except that an extra provision is made which allows traps in user mode to be handled through an alternate set of vectors, also located in kernel data space. Kernel and supervisor mode traps operate exactly as in the unmodified machine.

Unfortunately, it is necessary to send certain traps to the kernel regardless of the processor mode at the time of the trap, because the occurrence of certain types of traps are highly indicative of hardware malfunctions and hence should be handled by the kernel. Additionally, certain other types of traps (memory management traps, for instance) require additional actions which only the kernel is able to perform. [10]

All of the original hardware trap vectors (as opposed to the alternate hardware trap vectors available on the modified machine) must point to kernel routines since they

---

[10]. In the case of a memory management trap, the memory management unit becomes partially disabled and certain of the registers are frozen in the state existing at the time of the trap. The unit must be reenabled and the memory management status registers saved if the state of the processor at the time of the trap is to be properly reconstructed. Only the kernel is able to perform these actions since the memory management registers reside on the unibus and hence do not reside in the address space of any process. Only the kernel should perform these actions since memory management is one of the primary mechanisms by which isolation of processes is maintained.

will be invoked when traps occur in either supervisor or kernel mode. The kernel is coded in such a way that it never traps when operating properly. The occurrence of a kernel mode trap then is indicative of some catastrophic error and currently results in an immediate halt.

Since supervisor traps can not be sent directly to the supervisor (because supervisor and kernel traps share the same vectors), another mechanism is necessary to reflect supervisor traps back to the supervisor portion of the process. [11] The obvious, straight-forward solution would be to follow the example set by the hardware: simply push the old PC, PS pair on to the supervisor's stack, and then "return" to the supervisor portion of the process by executing an RTI. This cannot be done, however, if one wishes to avoid traps within the kernel. One can not mimic the hardware and merely push the old PC and PS onto the supervisor's stack, for there is no assurance that the stack portion of the supervisor's space will be core-resident and hence such an attempt could result in a

---

[11]. The same observation holds for interrupts as well. Traps and interrupts on the 11/45 are handled identically once generated. The difference between the two is that interrupts occur asynchronously and may be inhibited if the processor priority is maintained at a high enough level, while traps are synchronous in nature and cannot be inhibited. The synchronous nature of traps allows them to be handled in a more straightforward manner than interrupts. The method chosen for handling interrupts is discussed in the next section.

kernel memory management trap. Furthermore, there is no assurance that the supervisor's stack pointer will point at anything legal anyway. One is additionally faced with the question of where to reenter the supervisor portion of the process. If one wishes to avoid such problems, an alternative mechanism must be devised.

There is, however, one portion of the process's address space which our design guarantees to be core-resident at all times -- the shared kernel/process communications segment. An obvious place to save the old PC,PS at the time of the trap and any other information necessary to enable the process to recover from the trap is within that shared segment. Having thus provided an analog to the saving of the old PC,PS done by the hardware, the problem remains of imitating the change in the contents of the program counter which results from a real trap.

This was quite simple to accomplish once a decision was made concerning where the process should resume execution. Two obvious alternatives came to mind: 1) the process itself could somehow specify where it wished to be entered, or 2) the kernel could cause control to be passed to specific locations fixed for all processes.

The first alternative is closer to the environment offered by the real machine. However, it necessitates

additional overhead in the kernel since it requires kernel coding not only to allow the process to set these "trap vectors" but also requires additional table space in the kernel to keep track of them. Also, one would probably wish to provide a similar facility in the handling of user mode traps, so the amount of additional necessary space is doubled.

The second alternative might seem at first glance to be a more restrictive one, but in actuality is equivalent. If fixed locations are picked such that two words are available between each separate transfer location, then sufficient room exists for the process to place a jump instruction, so that the process can effectively set this pseudo-trap vector any way it wishes without causing any additional code or overhead in the kernel. The kernel then causes the supervisor to begin execution at a fixed location which contains an instruction to jump to a user-specified trap handler.

A similar problem exists with the alternate trap vectors that come into use when traps occur in user mode. The problem is not so critical here since there is no ambiguity concerning the processor mode at the time of the trap. If the kernel does not intervene, the hardware will properly deal with any additional traps which might result from attempting to push a PC,PS pair onto the process's

stack. The problem of where to reenter the process still exists, however, and unless one is prepared to save and restore the entire set of alternate trap vectors (1000 octal bytes!) each time processes are swapped, some other solution must be developed. [12]

Since it was already necessary to develop a pseudo-trap mechanism to deal successfully with supervisor traps, it seemed most straight-forward to employ a derivative of it to handle user traps. User traps which are not sent directly to the supervisor by the modified hardware are reflected back to the supervisor part of the process pair by the kernel in the same manner as supervisor traps except that the fixed entry points are different. [13] The supervisor can handle them with the same indirect jumping mechanism.

Supervisor traps cause the supervisor-mode portion of the process to be entered at the locations which would have

---

[12]. One could, of course, dedicate an additional 1000 octal bytes of kernel data space for each process (on well-chosen boundaries in physical memory) and reload the kernel relocation register which points to the alternate trap vectors each time a new process is loaded. This is a waste not only of real memory (for storing the alternate vector for the process) but also of the kernel's virtual memory: the remainder of the page frame containing the alternate vectors cannot be used for any "useful" kernel data unless an entire segment is reserved for each set of process trap vectors (and even then the usefulness of this data is questionable since its contents would change each time a new process is run).

contained the new PC, had the process been running on the real machine (i.e., a trap which would have used the contents of location 4 as the new PC causes the supervisor to be entered with its PC pointing at location 4 in the supervisor's instruction space. A trap to vector 10 will enter the supervisor at location 10, etc.).

Since no hardware 'trap' trap vectors exist above 40 octal in either the modified or unmodified 11/45 except for the memory management trap vector, real user traps (those caused by the user portion of the process pair) will cause the supervisor portion of the process to be entered at the normal trap vector location plus a bias of 40 octal bytes (i.e. a user vector 4 trap causes the supervisor to be entered at 44, a user vector 10 at 50, etc.). The previous user PC and PS are placed in locations in the shared segment reserved for the old user PC and PS, and the supervisor portion is entered with a real (supervisor) PS indicating current mode of supervisor, previous mode user

---

[13]. These are only the user traps which are sent directly to the kernel by the hardware. Currently memory management traps, user stack limit violations, memory parity traps, and power fails traps are fielded by the kernel and then reflected back to the supervisor portion of the process (the VMM). The alternate trap vectors are set by the kernel such that user bus-error traps, odd-addressing traps, privileged instruction traps, illegal and reserved instruction traps, and IOT, EMT, TRAP, and BPT traps are sent directly to the supervisor portion of the process and require no kernel intervention.

in order that the supervisor portion can access the user's virtual space without necessitating kernel intervention.

There is no confusion regarding whether the source of the trap was in the user or supervisor portion as long as the supervisor is careful to set up its pseudo trap vectors to point to different trap handlers for the two cases. [14] A user mode memory management trap causes the supervisor to be entered at location 100(octal), while supervisor memory management traps cause it to be entered at location 104 (octal).

---

[14]. These pseudo trap vectors effectively represent a branch table where each entry consists of two words (since a jump instruction on the 11/45 requires two words). Where on the bare machine the trap vectors might have looked like:

```
LOCATION  CONTENTS
0         VECTOR_0   % NEW PC
2         PS         % NEW PS
4         VECTOR_4   % NEW PC
6         PS         % NEW PS
10        VECTOR_10  % NEW PC
12        PS         % NEW PS
```

The equivalent pseudo trap vectors would look like:

```
LOCATION  CONTENTS
0         JMP        % JMP VECTOR_0
2         VECTOR_0
4         JMP        % JMP VECTOR_4
6         VECTOR_4
10        JMP        % JMP VECTOR_10
12        VECTOR_10
```

3.6 The Kernel Interrupt Handler

As long as one deals with I/O on the PDP 11/45 -- indeed, as long as one deals with I/O on most machines -- the problem of how to handle interrupts must be resolved. [15]

The infeasibility of kernel operation in complete polling mode necessitates that it operate in either full or partial interruption mode. As previously mentioned, however, it is inconvenient for the kernel to be arbitrarily interruptible. The PDP 11/45 hardware does, however, allow such issues to be taken in account: by

---

[15]. On the PDP 11/45, however, one could ignore interrupts entirely, simply by refusing to set the interrupt enable bit when I/O is initiated. One must then poll for device completion, however. While polling might be a reasonable choice in some systems, it is not a viable alternative if done by the controlling portion of a multi-programming system, such as the kernel. Two obvious problems arise if polling is done by the system controller, depending upon how often polling is done: 1) If the controller polls only when it has been entered in order to provide some process-requested action, then a delay occurs between the actual time of device completion and the time device completion is noticed which may have dire consequences, depending upon the timing characteristics of the particular device; 2) if the system controller polls whenever it has nothing else to do, then the situation is even worse since no multiprogramming occurs at all. There is a solution which lies halfway between the two mentioned above, but it requires use of at least one type of interrupt and the existence of some sort of clocking mechanism: the system controller arranges to receive clock interrupts periodically, polls for any device completions whenever it receives such a clock interrupt, and allows user processes to run whenever it has nothing else to do.

appropriate raising or lowering of the processor priority, one can selectively inhibit pending interrupts from occurring until later or allow them to occur. [16] Thus if one wishes not be be interrupted under any circumstances during critical sections of code, one merely raises the processor priority to its maximum level just before entering the critical section and then lowers the priority back to the appropriate level after the critical section of code has been executed. [17]

Any interruptibility problems the kernel might have had, then, are solved by appropriate use of the basic facilities provided by the hardware. Such is not the case for arbitrary processes, however. The 11/45 mode dependent restrictions prevent non-kernel mode programs from actually manipulating the processor priority unless by explicitly

---

[16]. The current implementation of the kernel does not make full use of the control over interruptibility offered by the hardware. The processor priority is set by the kernel such that the processor is totally uninterruptible whenever it is executing in kernel mode and arbitrarily interruptible whenever it is not in kernel mode (i.e. priority = 7 whenever the kernel is running, priority = 0 whenever the kernel is not running).

[17]. The 11/45 has an explicit, uninterruptible instruction, SPL (Set Priority Level), for manipulating the processor priority. The SPL instruction either NOPS or traps when executed in user or supervisor mode, depending upon how the virtual machine hardware modifications are set and whether they are enabled.

changing the hardware PS, and the kernel has removed the hardware PS from the address space of all processes. [18] But processes need as much control over their interruptibility as the kernel does. The real question is, how much control? Are "levels" of interruptibility such as offered by the hardware really necessary, or is a simple interruptible/non-interruptible distinction sufficient?

For "arbitrary" processes, the answer is unclear -- interruptibility levels might be crucial or irrelevant. Certainly some virtual machines may require levels of interruptibility since they were written for a base hardware which provides such facilities.

Whatever interrupt interface the kernel presents is not directed at "arbitrary" processes, however, but indeed at a decidedly unarbitrary half of an "arbitrary" process

---

[18]. It is necessary that processes be unable to change the processor priority; otherwise malicious processes could keep the kernel from promptly servicing interrupts destined for other processes or otherwise steal control of the machine by setting its priority to 7 and refusing to give up control. If a process were able to set its hardware processor priority to 7, then no assurance could be given that the kernel would ever again regain control of the machine. Currently that assurance is possible because 1) processes running in user or supervisor mode are prevented from changing the processor priority by the hardware, 2) the kernel returns control to user processes with the hardware processor priority set at level 0 (completely interruptible), and, 3) periodic clock interrupts force control of the machine to be relinquished to the kernel.

pair: while the user mode portion (VM) is arbitrary, the supervisor mode portion (VMM) is not; the supervisor portion is required to know about certain details of the kernel and its interface.

Hence the kernel is not constrained to provide all the features of the real hardware for the supervisor portion if it is easier or more convenient for the kernel to provide something simpler. How minimal can the kernel-provided features be with respect to interrupts? If the interruptible/uninterruptible distinction is sufficient for the kernel itself, then one suspects that it should suffice (or can be made to suffice) for the VMM as well.

Once the need for a pseudo-interrupt mechanism was recognized, it remained only to consider the possible alternatives, select the most acceptable one, and implement it.

Two issues are of importance with regard to interrupts: 1) whether to physically interrupt the process, and 2) how to prevent the process from losing interrupts which occur while it is in the process of transitioning

from a non-interruptible to interruptible state. [19]

One possible solution to the interruptiblity problem was to implement a pair of kernel calls: one to mark the process as uninterruptible, and one to mark the process as interruptible -- effectively a Disable-Interrupt kernel call and an Enable-Interrupt kernel call. This scheme would address the question of whether to interrupt the supervisor portion of the process and would perhaps make it more responsive to interrupts. The supervisor process would then have the ability to decide whether it wished to be interrupted or not. However, the second problem still remains if an interrupt occurs while the supervisor is uninterruptible, but on the verge of reenabling interrupts. Some additional mechanism is necessary to avoid losing this crucial interrupt.

---

[19]. An instance of this second problem occurs as follows:

```
DISABLE_INTERRUPTS;
X:IF NOTHING_LEFT_TO_DO
     THEN
          Y:ENABLE_INTERRUPTS;
          RETURN_TO_USER;
     ELSE
          DO_WHAT_NEEDS_TO_BE_DONE;
FI;
```

When the interrupt occurs between the successful test in statement X and the execution of statement Y. In the majority of cases, this merely results in the interrupt being delayed. However, if the VMM returns control to the VM and the VM thereafter never attempts any actions which cause the VMM to be invoked (attempts no sensitive or privileged actions), this interrupt is lost forever.

The solution offered by some systems is termed a "hyper-awake" mechanism and is typically implemented as follows: The process runs normally with interrupts enabled or disabled as desired. When it eventually decides it has nothing useful to do, it executes a "sleep" system call to inform the system that it cannot usefully run until it receives an interrupt. If an interrupt has occurred in the meantime, then the process becomes "hyper-awake" and the sleep request is ignored by the system and control returned to the calling process just as if the interrupt had happened after it had gone to sleep and not before. Solutions of this sort, however, typically require that the program exit (or sleep) by first returning control to the supervisor; if the program exits without first going through the supervisor, then the supervisor has no way of knowing it needs to force the program to be interrupted now. This would require that the VMM go through the kernel each time it returns to the VM, an action which occurs rather frequently. This, of course, requires some extra handling by the system supervisor (kernel).

When the VMM requests that the kernel perform I/O on the VMM's behalf, it can request that the kernel interrupt it when the I/O completes. If the VMM requests an interrupt from the kernel, it will receive its interrupt in one of two ways, depending on whether the VMM or the VM was

running when the interrupt happened.

Since it is often difficult to write code which is interruptible at any arbitrary point in time, the decision was made that the kernel would never forcibly interrupt the VMM--- that is it would never interrupt a process which was currently executing in supervisor mode.

This decision was not without problems, however. It fell prey to the lost interrupt problem sketched earlier. Accordingly, an additional mechanism was added.

The 'solution' provided for the lost interrupt problem is in several senses an unusual one. The mechanism is unintuitive yet it possesses the favorable quality of being extremely simple to implement within the kernel, while at the same time being fairly straight-forward to deal with in the VMM: When the kernel wishes to interrupt a VMM, it places an interrupt queue entry into the queue of the process to be interrupted. If the stored process status of the process to be interrupted indicates that the user portion of the process was active at the time of the interrupt [20], then the stored PC and PS are saved in

---

[20]. Here active is a relative term and is defined by the current-mode bits of the PS saved in the process table. If the current mode is user mode, then the user portion is active; otherwise the supervisor portion is active.

fixed spots of the communications segment (so that the user portion can be restarted from where it was interrupted, if desired), and the supervisor portion of the process caused to be entered at location 120 (octal) the next time the process is allowed to run. [21]

If the stored PS for the process indicates that the supervisor portion was running at the time of the interrupt, then instead of actually interrupting the supervisor portion, the kernel places a NOP instruction in the first word of the interrupt word portion of the communication segment.

When the supervisor portion of the process enters that portion of its code which handles the draining of the queue, it places several appropriate machine language instructions in successive words of the interrupt words portion of the communication segment, which when executed would cause the user to be reentered, a sleep call to be made, etc.-- whatever the VMM wants to do if it receives no further interrupts. In addition, it places in successive words of the kernel/process communications buffer whatever

---

[21]. This is accomplished quite simply by placing a supervisor PS in the stored PS slot in the kernel's process table and by placing 120 in the stored PC. The process PC and PS are loaded from the process table each time the process is run.

other instructions are necessary to cause it to be reentered in the proper spot for handling any additional interrupts. When the VMM finally tries to exit, it does so by jumping to the first word of the machine language instruction sequence it has placed in the communication segment. If the kernel has "interrupted" the VMM, then the first several words will have been nullified by being NOPed by the kernel and the VMM will be reentered wherever it wanted to be. If no additional interrupt has been given, then the VMM will exit as planned.

Thus, from the kernel's viewpoint the code to implement the pseudo-interrupt looks something like:

KERNEL CODE:

%%%

```
PROCEDURE INTERRUPT_HANDLER(OLD_PC,OLD_PS);
BEGIN
      % SAVE PROGRAM COUNTER OF PROCESS WHICH
      % WAS RUNNING
      % AT THE TIME OF THE INTERRUPT IN THE
      % KERNEL'S PROCESS
      % TABLE SLOT FOR THAT PROCESS
      PROCESS.PC[RUNNING_PROCESS] <- OLD_PC;

      % SAVE THE PROCESSOR STATUS
      % (CURRENT AND PREVIOUS MODES, CONDITION CODES)
      % OF THE PROCESS RUNNING AT TIME OF INTERRUPT IN
      % KERNEL'S TABLE FOR THAT
      % PROCESS
      PROCESS.PS[RUNNING_PROCESS] <- OLD_PS;

      % NOW FIGURE OUT WHICH PROCESS IS SUPPOSED
      % TO GET THE INTERRUPT
      P <- USER(DEVICE);
```

```
        % SEE IF PROCESS WANTS INTERRUPT
        IF INTERRUPT_REQUESTED(DEVICE)
             THEN
                  % PROCESS REQUESTED PSEUDO-INTERRUPT

                  % BUILD INTERRUPT INFORMATION TO
                  % SEND TO PROCESS
                  I <- BUILD_INTERRUPT_INFO(DEVICE);

                  % FINALLY SEND THE INTERRUPT
                  SEND_INTERRUPT(P,I);

        FI; % TO INTERRUPT_REQUESTED IF

        % NOW RETURN TO THE INTERRUPTED PROCESS
        % NOTE THAT THE PC,PS LOADED MAY BE
        % DIFFERENT FROM THE PC,PS
        % WHICH WAS SAVED UPON ENTRY
        % TO THE INTERRUPT HANDLER
        OLD_PC <- PROCESS.PC[RUNNING_PROCESS];

        OLD_PS <- PROCESS.PS[RUNNING_PROCESS];
    END; % OF INTERRUPT HANDLER
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    PROCEDURE SEND_INTERRUPT(PROCESS,INFORMATION);

    BEGIN

        % PROCESS IS PROCESS TO BE INTERRUPTED
        % INFORMATION IS INTERRUPT INFORMATION
        % TO BE PLACED INTO QUEUE

        % PUT INTERRUPT NOTIFICATION IN QUEUE
        PUSH_INTO_PROCESS_QUEUE(PROCESS,INFORMATION);

        % AND INTERRUPT THE PROCESS
        IF CURRENT_MODE(PROCESS.PS[PROCESS]) =
             SUPERVISOR_MODE

             THEN

                  % PROCESS IS CURRENTLY EXECUTING IN
                  % SUPERVISOR PORTION,
                  % SO DON'T INTERRUPT IT

                  % NOP FIRST WORD OF SUPERVISOR
                  % EXIT SEQUENCE CODE
                  % NOTE THAT PROCESS.INTERRUPT_WORD IS
```

```
                    % REALLY THE SAME AS
                    % COMMUNICATIONS.INTERRUPT_WORD
                    PROCESS.INTERRUPT_WORD[PROCESS,0] <-
                         NOP_INSTRUCTION;

               ELSE

                    % PROCESS IS CURRENTLY IN USER MODE
                    % FORCE IT INTO THE SUPERVISOR NEXT
                    % TIME THE PROCESS RUNS
                    % INTERRUPT ENTRY POINT FOR SUPERVISOR
                    % IS 120(OCTAL)
                    PROCESS.PC[PROCESS] <- 120K;

                    PROCESS.PS[PROCESS] <- SUPERVISOR_PS;

          FI; % TO CURRENT_MODE IF
     END; % OF SEND_INTERRUPT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Thus the code in the VMM might look something like:

```
SUPERVISOR CODE:

BEGIN
     LOCATION 120-122:
     % WHERE ENTERED IF INTERRUPTED WHILE
     % IN USER PORTION
     % THIS CODE TRANSFERS TO THE INTERRUPT HANDLER
     JMP INTERRUPT_HNDLR;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     % OTHER MISCELLANEOUS CODE AND DATA HERE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

     INTERRUPT_HNDLR:

     % END UP HERE WHEN A PSEUDO-INTERRUPT OCCURS

     % PUT RTI INSTRUCTION IN FIRST WORD OF
     % INTERRUPT SEQUENCE
     % THIS IS NOOP'ED BY KERNEL
     % IF ANOTHER INTERRUPT COMES IN
     % WHILE SUPERVISOR PORTION IS STILL RUNNING

     % WILL BE REENTERED HERE IF ANOTHER
     % INTERRUPT COMES IN BETWEEN TIME WE HAVE
     % DECIDED TO RETURN TO USER AND THE TIME
     % WE GET AROUND TO ACTUALLY DOING IT
     RE_DO:COMMUNICATIONS.INTERRUPT_WORD[0] <-
          RTI_INSTRUCTION;
```

```
% NOW FILL IN WHAT TO DO IF INTERRUPT COMES IN --
% I.E., IF KERNEL NOOP'S FIRST WORD
COMMUNICATIONS.INTERRUPT_WORD[1] <- JMP;
COMMUNICATIONS.INTERRUPT_WORD[2] <- RE_DO;

DRAIN_QUEUE:

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% CODE TO DRAIN ENTRIES OUT OF QUEUE GOES HERE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% PUT USER PS ON STACK SO CAN RTI TO USER PORTION
STACK_PUSH(USER_PS);

% PUT USER PC ON STACK FOR RTI
STACKPUSH(USER_PC);

% AND FINALLY ATTEMPT TO RETURN TO USER
% EITHER END UP AT RE_DO OR RETURN TO USER
% (THIS JUMPS INTO THE COMMUNICATIONS SEGMENT)
JMP @ADDRESS_OF_COMMUNICATIONS.INTERRUPT_WORD[0];

END; % OF SUPERVISOR CODE
```

One must admit that this design seems somewhat strange, clumsy, and unclean. [22] However, other schemes suggested, involving such things as enable and disable interrupt system calls, 1) are more complicated, 2) as system calls would require increased code in the kernel to support them, and 3) would be at least as, if not more, difficult to handle within the VMM.

---

[22]. One must realize that the process is now executing in a portion of the kernel's address space (since the communication segment is shared between kernel and supervisor) which might seem especially risky. This is not security relevant, however. The process inherits no special powers by doing so since privileges on the 11/45 are determined by a combination of process mode and relocation register contents and not merely a matter of where in memory the program counter points.

Thus the interruptible/non-interruptible flag is essentially contained in the stored PS of the process which is to receive the interrupt. A current mode of user implies that the process is interruptible, while a current mode of supervisor implies that is it not to be interrupted. The indirect jump return mechanism used by the VMM to return to the VM, when combined with the actions taken by the kernel when it places an interrupt in the process queue, effectively implements a drawn out uninterruptible "test and set" instruction.

3.7 The Kernel Call Handler

Entry into the call handling portion of the kernel is accomplished via the execution of an EMT instruction (emulator trap) while running in supervisor mode. [23]

Execution of the supervisor EMT instruction causes the kernel call handler to be entered with processor state of kernel mode, register set 0, and processor priority 7 (all interrupts inhibited).

The PC and PS of the running process are first saved, then the EMT is decoded. The kernel call-code is contained in the EMT instruction itself (so one executes a EMT 6 to do kernel call number 6, for instance). [24] The kernel decodes the call code to determine if the call is a legal one. If no such call code exists, no further action is taken by the kernel except that of returning error information to the calling process.

If, on the other hand, the call is at least superficially legal (i.e., the call code is within the

---

[23]. EMT's executed in user mode are totally transparent to the kernel since the alternate trap vectors are set to reflect user mode EMT's directly to the supervisor portion of the process.

[24]. This, of course, limits the number of possible distinguishable kernel calls to 256, since the EMT instruction contains only enough room for one byte of emt code.

range of valid kernel call codes), then the call arguments are copied from their locations in the shared kernel/process segment into local areas of the kernel's space. [25] All further actions occur relative to the values of these copied parameters. After the arguments are copied, control is passed to the kernel routine responsible for the implementation of the requested call.

---

[25]. No security flaws can result from this delayed copying that would not be possible even if the parameters were copied immediately. The kernel has restricted access to the communications segment such that it can never be the destination of any I/O transfer. Since I/O devices are the only entities in the system which operate asynchronously of the CPU, and since the kernel is in complete control of the CPU at the time of the call, it is impossible for the kernel call parameters to change between the time the kernel initially receives control and the time it copies the parameters into its own address space. Furthermore, even if it were possible for I/O to be directed into the communications segment, the very worst that could happen is that the kernel would make its security checks on parameters different from those initially given by the requesting process (or perhaps, think the process was requesting a different call). But this is only an issue if the security policy permits another process to write into the communications segment of the requesting process, a capability which in itself implies the requestor may be in plenty of trouble anyway.

The kernel could have automatically copied all possible kernel call parameter slots immediately, even before trying to decode the call, but no increased security is gained by doing so. If I/O were allowed to the communications segment, it is as likely to occur at the beginning of the call as it is to occur any other time.

3.8 Kernel Calls

A set of kernel calls provide the only visible interface between processes running under the kernel and the kernel. They are the means by which processes request the kernel to perform certain actions on their behalf, actions which can be performed only by the kernel.

A kernel call is made by placing appropriate information (kernel call arguments) in well-specified portions of the shared kernel/process communication buffer and then executing an EMT instruction. [26] The kernel then copies the information from the communications buffer into cells local to the kernel, decodes the call, and performs the requested action if it decides the request is a legal one, possibly returning error or other information back to the caller through locations in the communications buffer.

There are two obvious places where kernel call return information could be returned to the process: 1) in fixed locations in the shared communications buffer, and, 2) in the process queue. The first alternative is reasonable when the communication between the kernel and the process

---

[26]. There was no overpowering reason why EMT was chosen, other than some instruction which traps when executed in supervisor mode was necessary. TRAP, IOT, or BPT instructions would have worked just as well.

is synchronized. The second alternative is useful when the communication occurs asynchronously.

It might appear that all kernel call responses could be placed in fixed portions of the shared communications buffer in all cases, whether the caller was running or not. This, however, seemed unsatisfactory in the case where the kernel's response does not occur immediately, as in the case of the Request-I/O kernel call, where the kernel response occurs in two phases -- the response to the Request-I/O call and then when the I/O is finally started by the device scheduler's Start-Indirect-I/O call, when the validation of the I/O request is finally done.

It seemed only reasonable that any errors occuring from the eventual Start-I/O call be returned to the process that initially requested the I/O, since it is extremely unlikely that the device scheduler could have caused the error because it has only read access to the original I/O request. [27]

If the kernel's response to the Start-I/O call were to be placed in the kernel call response area of the original caller, then unusual and unpredictable results could occur.

---

[27]. It could, however, have attempted the Start-Indirect-I/O call while the device was still busy, which is a scheduling error on the scheduler's part and no fault of the requesting process at all. This error, however, is reflected back to the scheduler and not the original requestor.

It is conceivable that the caller might be logically in the middle of executing another kernel call, in which case the response from the earlier call would be confused with the response from the current kernel call. [28] Perhaps even more likely, the caller would have no reason to suspect that it had received information from the kernel. Of course the kernel could always 'wake-up' the process, but then the process would need to keep track of what it last saw as a kernel response or be forced to resort to some other equivalently complicated mechanism to decide what is transpiring.

The second alternative, that of placing all kernel call return entries in the process queue offers a solution to the problem which disqualified the first alternative, but it too possesses an annoying feature: the vast majority of kernel call returns occur synchronously and the necessity for searching the queue for the response to a

---

[28]. Because kernel calls typically require a number of instructions in order to move arguments into the communications buffer, it is reasonable to speak in terms of being "in the middle" of executing a kernel call, although, strictly speaking, it is physically impossible: either the EMT has already been executed, in which case the kernel will handle the call first and then overwrite the current call return with the response from the shared-I/O call previously initiated, or the EMT has not yet been executed, and the shared-I/O response is the one which will be overwritten by the response to the new call.

rather inconvenient for the VMM. [29]

A compromise between the two alternatives was implemented. If the kernel call response occurs asynchronously (as in the Start-Indirect-I/O response), it is placed in the queue. If the response occurs immediately (synchronously), it is placed in fixed locations in the communications buffer. This solution is only slightly more complicated than either of the previous alternatives, yet possesses neither of the drawbacks. The information needed by the process is placed where it can be most easily accessed at the time it is received.

A list of the designed kernel calls and a thumbnail sketch of their functions follows.

The CREATE-PROCESS kernel call is made to create a user-supervisor process pair.

The DESTROY-PROCESS kernel call is made to destroy a user-supervisor process pair.

---

[29]. Although the VMM could search the queue for the return information, that queue entry cannot in general be removed from the queue when found without endangering the information contained in the rest of the queue. The process queue is implemented in a manner where the kernel may freely modify its rear pointer and the VMM its front pointer. Thus pulling entries from the middle can create all sort of havoc for the VMM.

The START/STOP kernel call is made to start or stop an already existing process.

The INVOKE-PROCESS kernel call is made to switch control of the CPU to another process.

The SWAP-IN kernel call is made to swap a given disk segment into a given memory frame.

The SWAP-OUT kernel call is made to swap a given core segment into a given disk segment.

The ATTACH-SEGMENT kernel call is made to associate core segments with processes.

The RELEASE-SEGMENT kernel call is made to detach core segments from processes.

The CREATE-SEGMENT kernel call is made to create disk segments.

The DESTROY-SEGMENT kernel call is made to destroy disk segments.

The SLEEP kernel call is made to notify the CPU scheduler that the requesting process does not wish to run.

The SEND-MESSAGE kernel call is made to send a small message to another process.

The ATTACH-I/O-DEVICE kernel call is made to [illegible] I/O devices to processes.

The RELEASE-I/O-DEVICE kernel call is made to deallocate I/O devices from processes.

The START-I/O kernel call is made to initiate I/O to attached devices.

The STATUS-I/O kernel call is made to interrogate the status of attached I/O devices.

The REQUEST-I/O kernel call is made to request I/O on a shared device.

The START-INDIRECT-IO kernel call is made to start I/O on the behalf of another process.

3.8 Kernel Design Principles

As the design of the UCLA-VM system progressed, a number of design principles evolved. The first design principle, one present from the very beginning, was completely motivated by the verification goal: keep the kernel as small as possible. Anything that can possibly be removed from the kernel and forced outward into unproven code without jeopardizing protection and security should be pushed out of the kernel, even at the cost of complicating outer level processes.

Two previously mentioned examples of this principle in action were the decisions to remove process and device scheduling from the kernel, instead constructing scheduling processes running outside of the kernel. One might eventually wish to verify such schedulers, of course, but one might reasonably expect that layering them outside of the kernel rather than inside could not help but simplify the task.

An example of a function which by the primary design principle cannot be removed from the kernel is the actual physical loading of I/O device control registers. The necessity of kernel control over this operation becomes immediately clear once the realization is made that I/O devices on the PDP 11/45 operate completely independently

of any protection features the processor itself provides. I/O devices on the 11/45 use absolute memory addresses, not virtual ones and can, in general, access any location in the machine's memory. [30] If a process were allowed to load device control registers directly, the kernel could not guarantee either the privacy of data residing in memory or of data residing on external media.

A second design principle employed was the notion of least privilege, the policy of granting only the minimal set of capabilities necessary to enable the performance of a given function. If a process needs only "read" access to a section of memory in order to perform its proper function, there is no need to allow it "write" and "execute" access as well. This principle is of special use in the determination of the proper privileges to be granted to certain "special" processes in the system (such as CPU scheduler, disk scheduler, initiator, updater).

A third design principle that proved most useful was the assumption of maximum damage, the practice of worst case analysis. This was extremely useful in empirically deciding which functions must necessarily be performed by

---

[30]. Some I/O devices, however, are restricted to operation within the lower 32K of the machine's memory since they do not utilize the full 18 bit addressing necessary to access locations above the first 32K.

the kernel and which functions can be delegated to processes running outside of the kernel. Often envisioning a malevolent user process performing the function in question served to clarify the full extent of its security relevance.

A fourth principle applied to the design of the kernel was that of inserting run-time checks wherever possible immediately before the action to be performed - the notion of check-action pairs. If the security check is immediately followed by its corresponding action, the probability of the check becoming invalidated before the action contingent upon it is taken is significantly reduced.

Numerous security flaws in IBM OS/360 resulted from separating the security-relevant check from the security-relevant action, coupled with the implicit assumption at the time of the action that whatever had been checked remained valid at the time of the action. An excellent example of a security flaw of this kind is the case where system call parameters are allowed to remain in the address space of the calling process during the execution of the call, are first checked there for legality, then used later during the system call without being reverified. The security flaw occurs when an I/O operation previously initiated by the calling process

modifies the "checked" parameters between the time the parameters were validated and the time the parameters are actually used -- neatly circumventing the security checks entirely.

Careful attention was given both during design and implementation to keeping the security checks as close to their pertinent actions as possible, in a conscious attempt to avoid the sort of pitfall detailed above.

Chapter 4: Kernel Call Description

4.1 Kernel Call Description Overview

The following section of this thesis presents an overview of the function of each kernel call. [1] First, a general description of what the call does is given, then its parameters (if any) are enumerated and explained, along with any pertinent restrictions which the kernel places upon them. Finally, the actions taken by the kernel upon successful execution of the call are explained.

Following the English description, a Parnas-like, pseudo-procedural description is given in an attempt to capture the pertinent functional properties of each call. The notation employed does not conform to any one particular programming language, yet is reminiscent of PL/1, ALGOL 60, ALGOL 68, and various other block-structured languages and hence is probably self-evident for the most part. The only unobvious, perhaps, notation is that used for comments: comments are preceded with % (percent sign) and are terminated by the end of the textual line. Definitions of individual terms

---

[1]. These calls, of course, do not represent all actions taken by the kernel. Additional security relevant actions are taken in the kernel trap and interrupt handlers.

found in the Input/Output specifications are given in Appendix C.

The initial parameter of each call is the name of the requesting process. This parameter is not supplied by the caller, but is filled in by the kernel before executing the call.

One function and one procedure are used extensively in the Input/Output specifications and deserve mention here. The function is EVAL. EVAL is a general purpose routine which interrogates the kernel's policy protection data. EVAL(A,B,C) supplies a True/False answer to the question: Can A do B to C? The general purpose procedure used extensively in the Input/Output specifications is ERROR(R). ERROR(R) causes kernel call error return information to be placed in the appropriate place in the communications buffer of process R.

4.2 Kernel Initiation Primitives: Description

4.2.1 The Create-Process Kernel Call

The Create-Process kernel call is the means by which new processes come into existence. This call is generally restricted to the initiator, but need not necessarily be so.

The parameters of the Create-Process call are the name of the requesting process and the name of the process to be created. The name of the new process must be unique and conform to standard process name formats and conventions.

If the requesting process is allowed to create processes and the given process name is acceptable to the kernel, then a kernel process table entry is allocated to the new process, containing the name of the new process, with all relocation registers initialized as unassociated except for those reserved for kernel/process communication which are initialized to point at appropriate segments. [2] All other registers, including the starting program counter, are initialized to zero.

---

[2]. A minimum of three supervisor relocation registers are needed for the shared kernel/process communications segment: one portion must be read-only, one portion must be read/write, and one portion must be executable. This requires two D-space relocation registers and one I-space relocation registers.

Input-Output Specifications

```
CREATE-PROCESS(REQUESTOR,PROCESS);
      STRING REQUESTOR;
      % NAME OF REQUESTING PROCESS
      STRING PROCESS;
      % NAME OF PROCESS TO BE CREATED


FUNCTION:
IF EVAL(REQUESTOR,"CREATE-PROCESS",PROCESS)
      ANDIF LEGAL-PROCESS-NAME(PROCESS)
      THEN
            ALLOCATE-PROCESS(PROCESS);
      ELSE
            ERROR(REQUESTOR);
FI ;
```

4.2.2 The Destroy-Process Kernel Call

The Destroy-Process kernel call provides the mechanism by which processes are removed from the system. This call is generally restricted to use by the initiator.

The parameters of the Destroy-Process call are the name of the process requesting the action and the name of the process to be destroyed. If the requesting process is allowed to destroy the given process, if the given process is "stopped" (a Stop-Process kernel call has already halted the process), and if all I/O associated with the process to be destroyed has completed, then all devices and segments attached to the given process are "released" and the slot in the kernel's process table previously occupied by the process deallocated.

Input-Output Specifications

```
DESTROY-PROCESS(REQUESTOR,PROCESS);
      STRING REQUESTOR;
      % PROCESS NAME OF REQUESTOR
      STRING PROCESS;
      % NAME OF PROCESS TO BE DESTROYED


FUNCTION:
IF EVAL(REQUESTOR,"DESTROY-PROCESS",PROCESS)
      ANDIF STOPPED(PROCESS)
      ANDIF NO-IO-IN-PROGRESS(PROCESS)
      THEN
            DETACH-DEVICES(PROCESS);
            DETACH-SEGMENTS(PROCESS);
            DEALLOCATE-PROCESS(PROCESS);
      ELSE
            ERROR(REQUESTOR);
FI ;
```

4.2.3 The Stop/Start-Process Kernel Call

The Stop/Start-Process kernel call is the means by which the CPU scheduler becomes aware of the existence or non-existence of processes. [3] Its parameters are the name of the process which requested the call, the name of the process to be started or stopped, and a flag indicating whether the process is to be started or stopped. If the requestor is allowed to start or stop the process, then the CPU scheduler is given "Invoke Process" access to the process (in the case of a start) or the CPU scheduler's "Invoke Process" access to the process is revoked (in the case of a stop). [4]

---

[3]. This call may be superfluous. See the discussion in the section on the design of the Release-I/O-Device kernel call.

[4]. This is done by setting a bit which indicates whether the process can be run. It need not involve a change in the kernel's underlying policy protection data.

Input-Output Specifications

```
STOP-START-PROCESS(REQUESTOR,FLAG,PROCESS);
      STRING REQUESTOR;
      % NAME OF REQUESTING PROCESS
      STRING PROCESS;
      % NAME OF PROCESS TO BE STOPPED OR STARTED
      INTEGER FLAG;
      % WHETHER TO START OR STOP


FUNCTION:
IF EVAL(REQUESTOR,ACCESS,PROCESS)
      THEN
            IF ACCESS = "START-PROCESS"
                  THEN
                        STOPPED(PROCESS) <- FALSE;
                  ELSE
                        IF ACCESS = "STOP-PROCESS"
                              THEN
                                    STOPPED(PROCESS) <- TRUE ;
                              ELSE
                                    ERROR(REQUESTOR);
                        FI ;
            FI;
      ELSE
            ERROR(REQUESTOR);
FI;
```

4.3 Kernel Scheduling Primitives: Description

4.3.1 The Invoke-Process Kernel Call

The Invoke-Process kernel call is the mechanism by which control of the CPU is switched from one process to another. The calling process supplies to the kernel the name of the process to which it wishes to yield CPU control.

If the requesting process possesses the proper capabilities to be allowed to invoke the given process, then the process context (PC,PS, general registers, relocation registers, and stack pointers) of the requesting process are saved in the kernel's process table entry for the requesting process and the context of the given process loaded into the hardware from where it had last been saved in the kernel's process table.

Normally only the CPU scheduler possesses the invoke process capability, but this is only a matter of policy and thus subject to change. Also, since the capability checked is in the form of a triple, it is possible to allow a process to invoke certain other processes without giving it the capability to invoke all processes. [5]

It should be noted, perhaps, that once a process has given up its right of execution by 'invoking' another

process, it has no real assurance that it will ever regain execution control again. The only process guaranteed to regain control is the CPU scheduler, which is periodically invoked by the kernel in order to perform process scheduling.

---

[5]. This extra flexibility of control over invocation of processes is probably not really necessary. Other kernel calls, however, do require that access questions be stated in terms of triples, so if one's protection data evaluation procedure is structured in a general enough manner, one may be able to gain that added but perhaps unnecessary flexibility at no real extra cost.

Input-Output Specifications

```
INVOKE-PROCESS(REQUESTOR,PROCESS);
      STRING REQUESTOR;
      % PROCESS NAME OF REQUESTOR
      STRING PROCESS;
      % NAME OF PROCESS TO BE INVOKED


FUNCTION:
IF EVAL(REQUESTOR,"INVOKE-PROCESS",PROCESS)
      ANDIF STOPPED(PROCESS) = FALSE
      THEN
            SAVE-CONTEXT(REQUESTOR);
            LOAD-CONTEXT(PROCESS) ;
      ELSE
            ERROR(REQUESTOR);
FI ;
```

4.3.2 The Swap-In Kernel Call

The Swap-In kernel call is the means by which segments are moved from disk to core. Its parameters are the name of the process requesting the swap, the name of the segment to be swapped, and the memory frame location where the segment is to be placed.

If the requestor has "Swap In" access to the segment and the memory frame is free, then the memory frame is marked as not free and a swap in request constructed and sent to the disk scheduler. Eventual response is sent to the original requestor in much the same way as with normal shared I/O requests.

Input-Output Specifications

```
SWAP-IN(REQUESTOR,SEGMENT,MEMORY-FRAME);
      STRING REQUESTOR;
      % PROCESS NAME OF REQUESTOR
      STRING SEGMENT;
      % NAME OF SEGMENT TO BE SWAPPED IN
      INTEGER MEMORY-FRAME;
      % MEMORY FRAME WHERE SEGMENT IS TO BE PLACED


FUNCTION:
IF EVAL(REQUESTOR,"SWAP-IN",SEGMENT)
      ANDIF MEMORY-FRAME.FREE(MEMORY-FRAME)
      THEN
            MEMORY-FRAME.FREE(MEMORY-FRAME) <- FALSE ;
            SEND-SWAP-IN(REQUESTOR,SEGMENT,MEMORY-FRAME);
      ELSE
            ERROR(REQUESTOR);
FI ;
```

4.3.3 The Swap-Out Kernel Call

The Swap-Out kernel call is the means by which core segments are swapped out to disk. Its parameters are the name of the process requesting the swap and the name of the segment to be swapped, and the name of the disk to which it should be swapped. [6]

The requested segment must be currently in core and not locked and the requestor must possess "Swap Out" access to it. If these conditions are met, then the kernel's tables are updated such that only the disk scheduler has access to the segment, and a "swap out" request built and sent to the disk scheduler. Eventual response is reflected to the original requestor in much the same manner as in normal shared I/O operations.

---

[6]. This disk name may be redundant information, depending on what segment naming conventions are enforced. It is possible that the segment name itself fully determines the device (disk) to which it will be swapped.

Input-Output Specifications

```
SWAP-OUT(REQUESTOR,SEGMENT,DISK-NAME);
      STRING REQUESTOR;
      % PROCESS NAME OF REQUESTOR
      STRING SEGMENT;
      % NAME OF SEGMENT TO BE SWAPPED OUT
      STRING DISK-NAME;
      % NAME OF THE DISK THE SEGMENT IS TO BE SWAPPED TO


FUNCTION:
IF EVAL(REQUESTOR,"SWAP-OUT",SEGMENT)
      ANDIF IN-CORE(SEGMENT)
      ANDIF SEGMENT.LOCK-COUNT[SEGMENT] = 0
      THEN
            REMOVE-ALL-BUT-SWAP-ACCESS(SEGMENT);
            SEND-SWAP-OUT(REQUESTOR,SEGMENT,DISK-NAME);
      ELSE
            ERROR(REQUESTOR);
FI ;
```

4.3.2 The Swap-In Kernel Call

The Swap-In kernel call is the means by which segments are moved from disk to core. Its parameters are the name of the process requesting the swap, the name of the segment to be swapped, and the memory frame location where the segment is to be placed.

If the requestor has "Swap In" access to the segment and the memory frame is free, then the memory frame is marked as not free and a swap in request constructed and sent to the disk scheduler. Eventual response is sent to the original requestor in much the same way as with normal shared I/O requests.

4.4 Other Kernel Primitives: Description

4.4.1 The Attach-Segment Kernel Call

The Attach-Segment kernel call is the means by which processes running under the kernel gain direct access to in-core segments. [7] Due to the structure of the 11/45 memory management hardware, the maximum number of segments which can be accessed at any one time in such a fashion is limited to 32, since only the user and supervisor relocation registers are available for process use. [8]

The parameters of the Attach-Segment call are the name of the process requesting the attach, the name of the process to which the segment should be attached, the name of the segment to be attached, the relocation register name (number) with which the segment should be associated, and the access with which it should be attached (read, read/write, execute).

---

[7]. This access allows normal PDP 11/45 instructions to read or write data or execute instructions residing in the segment without intervention by the kernel. It is only after the successful execution of this call that the segment can be directly accessed.

[8]. Not all of the full range of segment accesses can be associated with each of the 32 available relocation registers. Only 16 of them can be attached with any "executable" access -- namely the I-space relocation registers.

If the requestor possesses "Indirect Attach Segment" access to the process to which the segment is to be attached, if the process to which the segment is to be attached possesses "Attach Segment" access to the requested segment with the requested access type, if the requested segment is core-resident [9] , and if the indicated relocation register of the process to which the segment is to be attached is currently unattached, then the indicated relocation register is set to point at the requested core segment with the requested access, the reference counter for the given segment is incremented, and the relocation register is marked as attached.

---

[9]. This restriction may be relaxed when (if) it is decided to allow non-resident segments to "fault" into memory when the process first tries to access a segment it has attached but which was not core-resident at the time of the attach.

Input-Output Specifications

```
ATTACH-SEGMENT(REQUESTOR,PROCESS-NAME,SEGMENT,ACCESS,
      RELOC-REG);
      STRING REQUESTOR;
      % PROCESS NAME OF REQUESTOR
      STRING SEGMENT;
      % NAME OF SEGMENT TO BE ATTACHED
      STRING PROCESS-NAME;
      % NAME OF PROCESS TO WHICH SEGMENT IS TO BE ATTACHED
      INTEGER ACCESS;
      % HOW SEGMENT SHOULD BE ATTACHED
      % (READ,READ/WRITE, ETC.)
      INTEGER RELOC-REG;
      % THE RELOCATION REGISTER WITH WHICH
      % THE SEGMENT IS TO BE ASSOCIATED


FUNCTION:
IF EVAL(REQUESTOR,"INDIRECT-ATTACH-SEGMENT",PROCESS-NAME)
      ANDIF EVAL(PROCESS-NAME,ACCESS,SEGMENT)
      ANDIF FREE(PROCESS-NAME,RELOC-REG)
      THEN
            FREE(PROCESS-NAME,RELOC-REG) <- FALSE ;
            RELOC-SET(PROCESS-NAME,SEGMENT,RELOC-REG,ACCESS);
            REF-COUNT(SEGMENT) <- REF-COUNT(SEGMENT) + 1;
      ELSE
            ERROR(REQUESTOR);
FI ;
```

4.4.2 The Release-Segment Kernel Call

The Release-Segment kernel call is the means by which core-resident segments become unassociated from process relocation registers. Once an attach segment call has associated a core segment with a given process's rel- register, it is necessary to release the segme- relocation register can be reused. [10]

The parameters of the Release-Segment name of the process requesting the release, the nam process from which the segment is to be released, name (number) of the relocation register unassociated. If the requestor is allowed to segment from the given process, if the given reloc register can be released, and if the relocation registe currently attached to a segment, then the relocati register is marked as unattached and the reference count for the previously attached segment decremented by one. [11]

---

[10]. This could have been done automatically in the Attach-Segment call, but would have complicated it somewhat. The code to release segments is needed anyway to free the segments attached to a process before it can be destroyed.

[11]. Some relocation registers, namely those associated with the shared kernel/process communications segment, effectively belong to the kernel and hence, for reasons related both to security and viability, cannot be released and subsequently reattached.

Input-Output Specifications

```
RELEASE-SEGMENT(REQUESTOR,RELOC-REG,PROCESS-NAME) ;
      STRING REQUESTOR;
      % PROCESS NAME OF REQUESTOR
      INTEGER RELOC-REG;
      % RELOCATION REGISTER TO WHICH SEGMENT
      % IS CURRENTLY ATTACHED
      STRING PROCESS-NAME;
      % PROCESS THE DEVICE IS TO BE RELEASED FROM


FUNCTION:
IF EVAL(REQUESTOR,"INDIRECT-RELEASE-SEGMENT",PROCESS-NAME)
      ANDIF FREE(REQUESTOR,RELOC-REG) = FALSE
      ANDIF RELOC-REG-CAN-BE-FREED(RELOC-REG)
      THEN
            REF-COUNT(SEGMENT(RELOC-REG)) <-
                  REF-COUNT(SEGMENT(RELOC-REG)) - 1;
            FREE(REQUESTOR,RELOC-REG) <- TRUE ;
            SEGMENT(RELOC-REG) <- "NULL";
      ELSE
            ERROR(REQUESTOR) ;
FI ;
```

4.4.3 The Create-Segment Kernel Call

The Create-Segment kernel call is the means by which processes create new disk segments. Its parameters are the process name of the requestor and the name of the segment to be created.

Create segment automatically gives the requestor (creator) attach segment access to the created segment. Create-Segment succeeds only if the requestor is allowed to create segments on the indicated disk, if the requested segment name is unique and conforms to standard kernel naming conventions and formats, and if there is sufficient space on the indicated disk for the segment to be allocated.

The name and location of the new segment are entered in the kernel's disk segment table, the newly created segment is zeroed, and the kernel's protection data is updated to show that the requestor now possesses "Attach-Segment" access to the segment.

Input-Output Specifications

```
CREATE-SEGMENT(REQUESTOR,DISK-NAME,SEGMENT-NAME);
      STRING REQUESTOR;
      % NAME OF PROCESS DOING THE CALL
      STRING DISK-NAME;
      % NAME OF THE DISK ON WHICH THE
      % SEGMENT IS TO BE ALLOCATED
      STRING SEGMENT-NAME;
      % NAME OF THE DISK SEGMENT TO BE CREATED


FUNCTION:
IF EVAL(REQUESTOR,"CREATE-SEGMENT",DISK-NAME)
      ANDIF ROOM-ON-DISK(DISK-NAME)
      ANDIF LEGAL-SEGMENT-NAME(SEGMENT-NAME)
      THEN
            ALLOCATE-DISK-SEGMENT(SEGMENT-NAME,DISK-NAME);
            ZERO-DISK-SEGMENT(SEGMENT-NAME,DISK-NAME) ;
            ADD-ACCESS(REQUESTOR,"ATTACH SEGMENT",
                  SEGMENT-NAME);
      ELSE
            ERROR(REQUESTOR);
FI ;
```

4.4.4 The Destroy-Segment Kernel Call

The Destroy-Segment kernel call is the only means by which disk segments are destroyed. Its parameters are the process name of the requestor and the name of the segment to be destroyed.

A segment may be destroyed only if the requesting process possesses "destroy segment" access to the segment and if no other process (or device) is using the segment.

Input-Output Specifications

```
DESTROY-SEGMENT(REQUESTOR,DISK-SEGMENT-NAME);
      STRING REQUESTOR;
      % PROCESS NAME OF REQUESTOR
      STRING SEGMENT-NAME;
      % NAME OF SEGMENT TO BE DESTROYED


FUNCTION:
IF EVAL(REQUESTOR,"DESTROY SEGMENT",SEGMENT-NAME)
      ANDIF REF-COUNT(SEGMENT-NAME) = 0
      THEN
            FREE-SEGMENT(SEGMENT-NAME);
      ELSE
            ERROR(REQUESTOR);
FI ;
```

4.4.5 The Sleep Kernel Call

The Sleep kernel call is the means by which processes tell the CPU scheduler that they do not wish to run for a while. Its only parameter is the process name of the requesting process. [12] Execution of the Sleep kernel call causes the general registers, relocation registers, process status, and program counter of the currently running process (the requesting process) to be saved, the CPU scheduler to be notified that the process does not wish to run [13], and the CPU scheduler to be invoked.

It is expected that the process which requested to sleep normally will not be run until it receives some interrupt or wakeup from the kernel, but this is dependent upon the coding of the CPU scheduler and hence should not be exclusively relied upon. The occurrence or non-occurrence of such a pending interrupt can be determined from examination of the queue of the requesting process.

---

[12]. It might be desirable to include an additional parameter, the amount of time the process wishes to sleep. This parameter would not be interpreted by the kernel, however, but would be sent to the CPU scheduler to be dealt with as desired.

[13]. This is done by re-setting the PROCESS-WANTS-TO-RUN bit associated with the requesting process.

The Sleep kernel call is similar in effect to an Invoke-Process call to invoke the CPU scheduler except that the Sleep call causes the CPU scheduler to be informed that the requesting process no longer desires to run. An additional difference is that it is possible that the requesting process might be able to invoke the CPU scheduler via a Sleep call which would not otherwise be able to do so by means of the normal invoke call. [14]

The main difference between the Sleep call and the Stop-Process call is that the latter causes the process which is being stopped to become non-invocable, thus insuring that the process will eventually quiesce enough to be destroyed.

In the Input/Output specifications which follow,

PROCESS-WANTS-TO-RUN(X,Y) <- Z,

causes the field associated with process Y in the array PROCESS-WANTS-TO-RUN for process X to be set to Z.

---

[14]. One might wish, for instance, to allow only the CPU scheduler to invoke other processes. Under such a policy, the sleep call would be the only way a process can keep itself from being run. This might seem of little importance, since it could simply continue executing in a tight "do nothing" loop instead, but this would not solve the problem if the process were really waiting for an interrupt to occur. If the process loops in the supervisor, the kernel will never forcibly interrupt it--hence the "do nothing" loop must in effect be a much more complicated "sleep looking for an interrupt entry in the queue" loop. This is mostly an efficiency issue.

Input-Output Specifications

```
SLEEP(REQUESTOR);
      STRING REQUESTOR;
      % REQUESTOR IS NAME OF PROCESS DOING THE CALL


FUNCTION:
BEGIN
      PROCESS-WANTS-TO-RUN(CPU-SCHEDULER,REQUESTOR) <-
            FALSE;
      SAVE-CONTEXT(REQUESTOR);
      LOAD-CONTEXT(CPU-SCHEDULER);
END;
```

4.4.6 The Send-Message Kernel Call

The Send-Message kernel call is the mechanism by which a process running under the kernel is allowed to communicate small units of information to other processes.

The parameters of the Send-Message kernel call are the process name of the requesting process, the name of the process to which the message is to be sent, and the message to be sent. If the requesting process is allowed to send messages to the destination process and if the message slot for messages from the requesting process to the destination process is empty, then the given message is placed in the message slot, the message slot is marked as full, and the destination process awakened.

These message slots are implemented as small, fixed size areas in the kernel/process communication segment of the receiving process. One slot is allocated for each of the possible N processes which are allowed to be active in the system at any moment in time. The flags which indicate whether a given message slot is full or not reside in a process-modifiable portion of the kernel/process communications segment. Hence a process can safe-guard itself against receiving messages from particular processes by insuring that the message slot full flag for that process always remains in the 'full' state. Since the

kernel automatically sets the full flag when it transmits a message, no old messages will be overwritten with new ones unless the receiving process resets the full flag.

Input-Output Specifications

```
SEND-MESSAGE(REQUESTOR,PROCESS,MESSAGE);
      STRING REQUESTOR;
      % REQUESTOR IS PROCESS NAME OF SENDER OF MESSAGE
      STRING PROCESS;
      % PROCESS IS PROCESS NAME OF RECEIVER OF MESSAGE
      WORD ARRAY MESSAGE;
      % MESSAGE TO BE SENT


FUNCTION:
IF EVAL(REQUESTOR,"SEND-MESSAGE-ACCESS",PROCESS)
      ANDIF MESSAGE.EMPTY(REQUESTOR,PROCESS) = TRUE
      THEN
            MESSAGE.INFORMATION(REQUESTOR,PROCESS) <-
                  MESSAGE;
            MESSAGE.EMPTY(REQUESTOR,PROCESS) <- FALSE ;
            WAKEUP(PROCESS);
      ELSE
            ERROR(REQUESTOR);
FI ;
```

4.5 Kernel I/O Primitives: Description

4.5.1 The Attach-I/O-Device Kernel Call

The Attach-I/O-Device kernel call is the means by which a dedicated I/O device becomes allocated to a process running under the kernel. A process which has attached a dedicated device is then allowed to issue start I/O requests for that device, receive status information from the device (do a Status-I/O call on it), and release it. A process cannot perform a successful Start-I/O kernel call on a device to which it is not attached.

The parameters for the Attach-I/O-Device call are the name of the process wishing to attach the device, the name of the process to which the device should be attached, and the name of the device it wishes to attach.

If the requestor is allowed to attach the device to the given process, if the given process is allowed to have the device attached to it, and if either the device is currently unattached or if the device is currently attached to the requesting process and is currently inactive (not involved in any ongoing I/O), then the device status is zeroed [15], the device is marked as attached to the given process, the given process becomes the owner of the device,

thereby inheriting "Start I/O", "Status I/O", and "Release I/O Device" access to the device [16], and the given process is informed that it now possesses the given device. If the device was attached to the requesting process at the time of the call, then the device status is zeroed, the device becomes detached (released) from the requesting process and the requesting process notified that it no longer "owns" the device. [17]

---

[15]. The device status is zeroed for precisely the same reasons as segments are zeroed when they are created -- to prevent the passage of unauthorized information. For most devices, zeroing status implies the pulsing of a special bit which causes the device registers to be reset to the initial state which results from the execution of a RESET instruction. Other devices have no device reset bit, but will be set to some well-defined, uninformative state.

[16]. This inheriting of privileges results from marking the requesting process as the owner of the device, thus permitting the process to perform start I/O, status I/O, and release I/O device kernel calls on the device.

[17]. The process to which the device has become attached will receive a "wakeup" from the kernel, in order to give it a chance to "notice" that it has been given a new device. This is of particular importance in insuring that the initiator can properly deal with a terminal which is being re-attached to it after the process which previously owned it decides to go away. The requesting process will receive no "wakeup", however. Presumably this is no inconvenience since the process effectively asked the kernel to remove the device.

Input-Output Specifications

```
ATTACH-IO-DEVICE(REQUESTOR,PROCESS,DEVICE);
      STRING REQUESTOR,PROCESS,DEVICE;
      % REQUESTOR IS PROCESS NAME OF CALLER
      % PROCESS IS NAME OF PROCESS TO RECEIVE DEVICE
      % DEVICE IS THE NAME OF DEVICE TO BE ATTACHED


FUNCTION:
IF EVAL(REQUESTOR,"ATTACH-IO-DEVICE",PROCESS)
      ANDIF EVAL(PROCESS,"ATTACH-DEVICE",DEVICE)
      THEN
            IF OWNER(DEVICE) = "NULL"
                  THEN
                        ZERO-DEVICE-STATUS(DEVICE);
                        OWNER(DEVICE) <- PROCESS ;
                        ATTACHED(DEVICE,PROCESS) <- TRUE ;
                        WAKEUP(PROCESS);
                  ELSE
                        IF OWNER(DEVICE) = REQUESTOR
                              ANDIF DEVICE-INACTIVE(DEVICE)
                              THEN
                                    ZERO-DEVICE-STATUS(DEVICE);
                                    ATTACHED(DEVICE,REQUESTOR) <-
                                          FALSE ;
                                    OWNER(DEVICE) <- PROCESS;
                                    ATTACHED(DEVICE,PROCESS) <-
                                          TRUE ;
                                    WAKEUP(PROCESS);
                              ELSE
                                    ERROR(REQUESTOR)
                        FI;
            FI ;
      ELSE
            ERROR(REQUESTOR)
FI ;
```

4.5.2 The Release-I/O-Device Kernel Call

The Release-I/O-Device kernel call is the means by which processes running under the kernel deallocate I/O devices they have been using. Due to security constraints, no I/O device may be released while I/O concerning it is active.

The parameters of the Release-I/O-Device call are the name of the process wishing to release the device and the name of the device to be released. If the requestor is attached to the device and the device is not busy (does not have I/O in progress), then the device is marked free (unattached) and the requestor's "Start I/O", "Status I/O", and "Release I/O Device" access to the given device revoked. [18]

---

[18]. This is done by marking the device as belonging to the "null" process, which is equivalent to stating it has no owner. Start-I/O, Status-I/O, and Release-I/O all require that the requestor "own" the given device.

Input-Output Specifications

```
RELEASE-IO-DEVICE(REQUESTOR,DEVICE);
      STRING REQUESTOR;
      % PROCESS NAME OF REQUESTOR
      STRING DEVICE;
      % NAME OF DEVICE TO BE RELEASED

FUNCTION:
IF OWNER(DEVICE)= REQUESTOR
      ANDIF BUSY(DEVICE) = FALSE
      THEN
            OWNER(DEVICE) <- "NULL";
            ATTACHED(DEVICE,REQUESTOR) <- FALSE;
      ELSE
            ERROR(REQUESTOR) ;
FI ;
```

4.5.3 The Start-I/O Kernel Call

The Start-I/O kernel call is the mechanism by which processes running under the kernel cause the kernel to initiate I/O in their behalf.

The parameters of the Start-I/O kernel call are the name of the requesting process, the name of the device to be started, the name of the core segment to be involved in the transfer, the byte offset from the beginning of the core segment where the I/O is to start, the number of bytes to be transferred between the core segment and the device, the access type to be performed in this I/O operation (read,write,rewind,etc.), whether a pseudo-interrupt is desired upon completion of the I/O, and other miscellaneous device-dependent information which is necessary for some devices, such as a disk segment name and offset for disk I/O requests.

If the given device is attached to the requesting process with the appropriate access, if the requesting process is allowed to access the given core segment with the appropriate access, if the named core segment is indeed core-resident, if the combination of segment byte offset and byte count indicate that the requested I/O will not extend beyond the named core segment, and if all other device-dependent checks indicate that the given request

will not violate any security constraints, then the kernel locks the core segment if necessary [19], marks the segment as being involved in ongoing I/O, and starts the device as requested.

Supplementary, device-dependent information may be returned to the requesting process along with some indication of the success of the call.

---

[19]. Some I/O operations, for instance input operations from terminals, do not require the locking of the core segment since all transfers to or from the core segment will be complete before the kernel returns control to the requesting process.

Input-Output Specifications

```
START-IO(REQUESTOR,DEVICE,SEGMENT,BYTE-OFFSET,BYTE-COUNT,
     OPERATION,INTERRUPT-FLAG,REQUEST);
     STRING REQUESTOR;
     % PROCESS NAME OF REQUESTOR
     STRING DEVICE;
     % NAME OF DEVICE INVOLVED IN I/O
     STRING SEGMENT;
     % NAME OF CORE SEGMENT INVOLVED IN TRANSFER
     INTEGER BYTE-OFFSET;
     % BYTE-OFFSET FROM BEGINNING OF CORE SEGMENT WHERE
     % TRANSFER WILL START
     INTEGER BYTE-COUNT;
     % NUMBER OF BYTES TO BE TRANSFERRED
     INTEGER OPERATION;
     % WHAT SORT OF OPERATION IS REQUESTED
     % (READ,WRITE,ETC.)
     BOOLEAN INTERRUPT-FLAG;
     % WHETHER PSEUDO-INTERRUPT WANTED
     WORD ARRAY REQUEST;
     % OTHER DEVICE-DEPENDENT INFORMATION


FUNCTION:
IF OWNER(DEVICE) = REQUESTOR
     ANDIF EVAL(REQUESTOR,OPERATION,SEGMENT)
     ANDIF IN-CORE(SEGMENT)
     ANDIF IN-SEGMENT-BOUNDS(BYTE-COUNT,BYTE-OFFSET)
     ANDIF LEGAL(DEVICE,OPERATION,BYTE-COUNT,BYTE-OFFSET,
          SEGMENT,REQUEST)
     ANDIF (SEGMENT.CAN_BE_USED[SEGMENT] = TRUE
          ORIF (SEGMENT.CAN_BE_USED[SEGMENT] = FALSE
               ANDIF (OPERATION = "SWAP-IN"
                    ORIF OPERATION = "SWAP-OUT" )))
     THEN
          DEVICE-START(REQUESTOR,DEVICE,SEGMENT,BYTE-OFFSET,
               BYTE-COUNT,OPERATION,INTERRUPT-FLAG,REQUEST);
          BUSY(DEVICE) <- TRUE ;
     ELSE
          ERROR(REQUESTOR) ;
FI ;
```

4.5.4 The Status-I/O Kernel Call

The Status-I/O kernel call is the mechanism by which a process can interrogate the status of I/O devices it has attached and effectively monitor the progress of any on-going I/O.

Its parameters are the name of the requesting process and the name of the device for which status is being requested. If the requestor currently has the device attached, then status information is returned to the requestor.

Input-Output Specifications

```
STATUS-IO(REQUESTOR,DEVICE,ACCESS);
      STRING REQUESTOR;
      % PROCESS NAME OF REQUESTOR
      STRING DEVICE;
      % NAME OF DEVICE WHOSE STATUS IS TO BE INTERROGATED
      INTEGER ACCESS;
      % WHAT STATUS IS REQUESTED


FUNCTION:
IF OWNER(DEVICE) = REQUESTOR
      THEN
            SEND-DEVICE-STATUS(DEVICE,ACCESS,REQUESTOR);
      ELSE
            ERROR(REQUESTOR);
FI ;
```

4.6 Kernel Shared I/O Primitives: Description

4.6.1 The Request-I/O Kernel Call

The Request-I/O kernel call is the mechanism by which processes request I/O be performed on their behalf by the shared device scheduler associated with the appropriate shared device. (such as the disk). Its parameters are the name of the requesting process and a Start-I/O request.

If the requestor is allowed to use the shared device and if the requestor has no other pending I/O requests for that shared device, then the request is placed in a "locked box", the appropriate device scheduler given access to the request, and the appropriate device scheduler is informed that it has a pending I/O request from the requesting process. [20]

---

[20]. A restriction imposed by the kernel regarding use of shared devices prohibits individual processes from having more than one pending request per shared device. This restriction simplifies the handling of locked boxes significantly and is also necessary in order to insure that device schedulers remain information sinks. The reader is referred to a paper by Popek and Kline [POPEK74A] for further discussion the information sink problem.

Input-Output Specifications

```
REQUEST-IO(REQUESTOR,DEVICE,SEGMENT,BYTE-OFFSET,BYTE-COUNT,
      ACCESS,INTERRUPT-FLAG,REQUEST);
      STRING REQUESTOR;
      % PROCESS NAME OF REQUESTOR
      STRING DEVICE;
      % NAME OF DEVICE INVOLVED IN I/O
      STRING SEGMENT;
      % NAME OF CORE SEGMENT INVOLVED IN TRANSFER
      INTEGER BYTE-OFFSET;
      % BYTE-OFFSET FROM BEGINNING OF CORE SEGMENT WHERE
      % TRANSFER WILL START
      INTEGER BYTE-COUNT;
      % NUMBER OF BYTES TO BE TRANSFERRED
      INTEGER ACCESS;
      % WHAT SORT OF OPERATION IS REQUESTED
      % (READ,WRITE,ETC.)
      BOOLEAN INTERRUPT-FLAG;
      % WHETHER PSEUDO-INTERRUPT WANTED
      WORD ARRAY REQUEST;
      % OTHER DEVICE-DEPENDENT INFORMATION

FUNCTION:
IF EVAL(REQUESTOR,"REQUEST-IO",DEVICE)
      ANDIF LOCKED-BOX.FREE[
            LOCKED-BOX-INDEX(REQUESTOR,OWNER(DEVICE)) ]
      THEN
            BOX-INDEX <-
                  LOCKED-BOX-INDEX(REQUESTOR,OWNER(DEVICE));
            LOCKED-BOX.REQUESTOR[BOX-INDEX]<-
                  REQUESTOR;
            LOCKED-BOX.DEVICE[BOX-INDEX]<-
                  DEVICE;
            LOCKED-BOX.SEGMENT[BOX-INDEX]<-
                  SEGMENT;
            LOCKED-BOX.BYTE-OFFSET[BOX-INDEX]<-
                  BYTE-OFFSET;
            LOCKED-BOX.BYTE-COUNT[BOX-INDEX]<-
                  BYTE-COUNT;
            LOCKED-BOX.ACCESS[BOX-INDEX]<-
                  ACCESS;
            LOCKED-BOX.INTERRUPT-FLAG[BOX-INDEX]<-
                  INTERRUPT-FLAG;
            LOCKED-BOX.REQUEST[BOX-INDEX]<-
                  REQUEST;
            WAKEUP(OWNER(DEVICE));
      ELSE
            ERROR(REQUESTOR);
FI;
```

4.6.2 The Start-Indirect-I/O Kernel Call

The Start-Indirect-I/O kernel call is the means by which device schedulers initiate I/O requests for processes which do not normally possess full "Start I/O" access to a shared device. Its parameters are the name of the device scheduler process and the "locked box" which contains the I/O request to be initiated.

If the indicated locked-box is not empty (not free) and if the requestor is the owner of the device for which the I/O request contained in the locked-box is intended, then the process whose name is contained in the "requestor" field of the locked box (the original requestor) is temporarily given "Start I/O" access to the device [21], and an internal Start-I/O call performed, with the name of the locked box owner as its requestor. [22]

This call returns two responses, one to the original requestor regarding the results of the Start-I/O call, and one to the device scheduler concerning the results of the Start-Indirect-I/O call. Interrupt notification is always sent to the device scheduler, but is sent to the original requestor only if so requested in the original Request-I/O

---

[21]. This is done by temporarily making the original requestor the owner of the device. Once the device is started, the owner is reset to to be the process which requested the start-indirect I/O call.

call.

---

[22]. Hence all the security checks will be made according to the capabilities of the original requestor. This may seem a bit unusual, and one might be tempted to "assume" certain properties of device schedulers and instead give them the pertinent capabilities at least temporarily. However, in the case where the original requestor attempted a syntactically or semantically invalid I/O request, one would like the error to be reflected back to the requestor and not the scheduler. If the scheduler instead receives the error, it is then faced with the problem of reflecting the error back to the requestor, which it can not do if it is to remain an information sink.

Input-Output Specifications

```
START-INDIRECT-IO(REQUESTOR,BOX-INDEX);
     STRING REQUESTOR;
     % REQUESTOR IS PROCESS NAME OF THE DEVICE SCHEDULER
     INTEGER BOX-INDEX;
     % LOCKED BOX INDEX RELATIVE TO THE DEVICE SCHEDULER


FUNCTION:
IF LOCKED-BOX.FREE[BOX-INDEX] = FALSE
     ANDIF OWNER(LOCKED-BOX.DEVICE[BOX-INDEX]) =
          REQUESTOR
     ANDIF BUSY(LOCKED-BOX.DEVICE[BOX-INDEX]) =
          FALSE
     THEN
          OWNER(LOCKED-BOX.DEVICE[BOX-INDEX]) <-
               LOCKED-BOX.REQUESTOR[BOX-INDEX];
          START-IO(LOCKED-BOX.REQUESTOR[BOX-INDEX],
               LOCKED-BOX.DEVICE[BOX-INDEX],
               LOCKED-BOX.SEGMENT[BOX-INDEX],
               LOCKED-BOX.BYTE-OFFSET[BOX-INDEX],
               LOCKED-BOX.BYTE-COUNT[BOX-INDEX],
               LOCKED-BOX.ACCESS[BOX-INDEX],
               LOCKED-BOX.INTERRUPT[BOX-INDEX],
               LOCKED-BOX.REQUEST[BOX-INDEX]);
          OWNER(LOCKED-BOX.DEVICE[BOX-INDEX]) <-
               REQUESTOR;
          LOCKED-BOX.FREE[BOX-INDEX] <- TRUE ;
     ELSE
          ERROR(REQUESTOR);
FI ;
```

Chapter 5: Kernel Call Design Issues

5.1 Kernel Call Design Overview

The following section is concerned with the detailed design of the kernel calls. Calls are presented in turn, along with any relevant design problems, decisions, or unusual features that arose relative to the particular calls. Sections concerning some calls are absent. The fact that a design discussion of a particular kernel call is not included here does not necessarily imply that the particular call is lacking in interest from the design point of view. Quite the contrary, some of the missing design discussions are related to calls which are of great interest, but whose design is interleaved and dependent to a great extent on the logical and physical representation of the policy protection data (which itself is yet to be designed).

An attempt was made whenever possible to indicate what influence each call had upon the general system structure as well as upon other kernel calls.

5.2 Kernel Initiation Primitives: Design

5.2.1 Create-Process Design

One important design problem of the Create-Process kernel call concerned the question of how the CPU scheduler becomes aware of the existence and name of the newly created process. The general overriding problem of queue overflow again arises with respect to the CPU scheduler. Simply stated, the problem is this: the CPU scheduler needs to be informed of the identity of processes as they come into existence, in order to be able to invoke them. Four obvious alternative exist: 1) processes are invoked by process number rather than by process name, 2) a kernel call is provided which returns the process name associated with a given process number, 3) the kernel tells the CPU scheduler the name of the new process when the process is created, or 4) the CPU scheduler is allowed read access to certain areas of the kernel's data structures which contain process name-process number associations and knows how to interpret the data structures.

The fourth alternative is unacceptable in the sense that it conflicts with the notion of least privilege, and more importantly, it unreasonably binds CPU scheduler coding to that of the kernel, restricting the ability to make perhaps necessary changes in the physical format of

kernel data structures without necessitating corresponding CPU scheduler modifications.

The third alternative again poses the question of where to place this asynchronous information. Several alternatives exist: 1) in the queue, possibly causing queue overflow problems; 2) in a fixed spot in the communications buffer -- information so placed, however, is likely to be lost if the CPU scheduler is not run quickly or often enough; or, 3) sent as a message from the initiator to the CPU scheduler via the send-message facility.

The second alternative is somewhat more attractive. It solves the problem in a "clean" way, yet it does necessitate an additional kernel call. It might present a bad precedent -- one of writing a new kernel call to solve any problem that might develop. Hence it might be well to consider other reasonable alternatives before choosing the kernel call route.

The first alternative, that of using process numbers instead of process names in the invoke process call, would at first glance seem to invclve the least number of drastic repercussions. It does, however, mean that the security implications of using process numbers rather than names in invoke process must be considered.

The question then is, are there any serious security drawbacks which would result from using process numbers as invoke process arguments rather than process names? As long as the kernel maintains the association between process names and numbers, there are no kernel security problems as far as the call is concerned , since the kernel can always use the corresponding process name in all of its security checks. The problem, if any, would seem to be one of whether any security assurances are lost for the CPU scheduler if numbers now replace names.

Consider the following scenario: The CPU scheduler decides that after running process X it will run process 2, a high-priority process, but between the time it invokes process X and again receives control, process 2 has been destroyed and another process created which is reassigned as process 2. The CPU scheduler receives control again and this time invokes the bogus process 2, which is not really the high priority process the scheduler thought it was. The CPU scheduler has been fooled into running a process it did not mean to.

Is such a scenario realistic (or even possible)? The answer seems to be, maybe. First, since only the initiator has the ability to create and destroy processes, clearly the initiator must have run sometime between the time it invoked process X and the time it attempted to invoke

process 2. Since only the CPU scheduler can cause processes to be run, it follows that process X itself must have been the initiator. Since the CPU scheduler invoked process 2 directly after process X, the initiator must have both destroyed the old process 2 and created the new process 2 before the scheduler again received control. The CPU scheduler was presumably given notification of both the demise of the old process 2 and the advent of the new process 2.

Now, either the new process 2 is or is not runnable. If it is not, then presumably the scheduler will suspect something is awry when the invoke fails and straighten things out without doing any real damage. If the new process 2 is runnable, then the scheduler has been effectively subverted, unless the notification given the scheduler concerning the destruction of process 2 and the creation of the new process 2 is sufficient to suggest to the the CPU scheduler that process 2 is now indeed a new entity.

So it would seem that the switch to process numbers is a reasonable one, provided that the details of process creation and destruction notification are carefully designed and considered with the invocation problem in mind.

There is, however, yet another viability issue at hand here. If the CPU scheduler is to be given the ability to do any sort of priority based scheduling, it needs some reasonable criteria upon which to base its scheduling decisions. If process numbers are now used instead of names, then one of the prime candidates, the process name itself, for information upon which priority can be determined, is now removed from the possibilities. This is not a significant restriction, however, if one allows the process creator to communicate with the CPU scheduler via the Send-Message call. Any priority-associated information lost by switching to process numbers can be easily regained and augmented through use of Send-Message.

5.2.2 Destroy Process Design

Most of the unusual constraints involved in the Destroy-Process kernel call are directly related to two observations: 1) terminating an arbitrary I/O operation in the middle is in general a difficult task, and 2) I/O once initiated will eventually terminate after some finite period of time -- it will not continue indefinitely.

The first observation makes it desirable that the kernel never forcibly attempt to terminate an I/O operation once it is started. The second observation lends credibility to the viability of never making such an attempt to terminate I/O abruptly.

One might reasonably ask why the kernel would ever need to stop an I/O anyway. The answer is obvious in retrospect: if one wishes to destroy a process, one must also terminate all on-going I/O before removing the process from the system; otherwise, the I/O in progress is liable to encroach upon the security of whatever data may happen to end up in the memory previously occupied by portions of the destroyed process.

5.2.3 Start/Stop-Process Design

The design and necessity for the Start/Stop-Process kernel call is heavily dependent upon the design and implementation details of the Invoke-Process, Release-I/O-Device, Attach-I/O-Device, and Sleep kernel calls. Additional discussion of the Start/Stop-Process call can be found in the design discussions of these calls. It is, however, appropriate to gather the pertinent facts together in one place.

Briefly stated, the motivations for the Start/Stop-Process call were: 1) the CPU scheduler needed some way of knowing when a new process came into existence or was destroyed; 2) some mechanism whereby I/O devices (terminals in particular) could reliably be removed from malfunctioning processes was desired; and, 3) some mechanism was desired whereby the CPU scheduler could be prevented from invoking a given process until certain resources had been allocated to the process. The Start/Stop-Process call was designed to meet these needs.

Through clever manipulation of the "invocable/non-invocable" bit which the Start/Stop-Process call controls (the "stopped" bit), a process (the initiator, for instance) can prevent the CPU scheduler from running the given process until all necessary resources

have been allocated to the process. The CPU scheduler learns of the existence of the new process either by interrogating the "stopped" bit -- either directly if it is able to read the bit, or indirectly by attempting to invoke the process and either succeeding or failing. One can insure that I/O devices can eventually be reclaimed by stopping the malfunctioning process and waiting until any I/O in progress completes, with the assurance that the process will not be able to "sneak in" and start any additonal I/O in the meantime.

All of the above functions, however, are related to system viability and are not data security relevant. Consequently this call is a likely candidate for removal from the kernel, in spite of its great simplicity.

5.3 Kernel Scheduling Primitives: Design

5.3.1 Invoke-Process Design

Two issues of interest arise with regard to the Invoke-Process kernel call besides the obvious policy issue concerning which processes should be allowed to make the call. The first is concerned with the conditions under which an invoke may be invalid, even for processes which normally possess the full invoke process capability (such as the CPU scheduler).

One obvious constraint is that the process which is to be invoked must indeed be an existing process. However, this simple constraint is not quite strong enough to capture the full extent of process scheduling requirements: There are instances during the lifetime of a process during which it is crucial that the process not be allowed to run (while the initiator is trying to destroy it or trying to recover a misbehaving device from it, for instance). Additionally, if one later wished to place the responsibility for swapping into a special swapping process rather than with the CPU scheduler, one might be interested in insuring that the CPU scheduler never run a process which does not have all of its necessary segments in core. The mere process existence assertion assured by the initial constraint offers nothing useful in either of the

preceeding cases. Something stronger must be asserted.

Exactly how much stronger must the constraint be? It seems fairly clear that a simple invocable/non-invocable switch should suffice: it is either "safe" for the CPU scheduler to run a given process or it is not. As long as the "proper" processes are able to manipulate this switch (where "proper" is something that must be contextually determined, according to what sort of guarantees one hopes to offer by the ability to prevent a process from being run), there should be no problem.

Of course it would be quite easy to effectively strangle the system by unwise use of such a blocking capability by a process which possesses it. Even this powerful ability can be tempered, however, by appropriate choice of policies: one might give a reliable system process (the initiator, for instance) the capability to stop any process, but limit this capability with regard to all other processes so that they can not stop the initiator. If one additionally gives the initiator the unrestricted ability to "unblock" processes, then one effectively has guaranteed that any system block-ups which may result can at least be undone by a reasonably clever initiator. One would not, however, wish to give the unblock capability to the CPU scheduler or all the nice guarantees just provided fly right out the window.

The second issue relates directly to issues discussed to some extent in the earlier section on Create-Process design issues and concerns the method by which one refers to the process to be invoked. There seems to be some slight advantage in invoking processes by name rather than by number, but only in the sense that one is slightly more sure that the process which one has invoked was the same process which one wished to invoke. It is not at all clear, however, how much has been lost if the process actually invoked is not the same process as the one conceptually invoked. The additional complications which using process names introduce in other kernel calls tend to overshadow any vague misgivings one might have about accidentally invoking the "wrong" process.

There is a definite trade-off here between names and numbers, but it appears that process numbers are substantially more convenient in terms of kernel manipulation and can easily be processed into their corresponding process names whenever kernel security checks require the actual names. [1]

5.3.2 Swap-In/Swap-Out Design

As mentioned earlier in the section concerning memory management, the data security constraints which are relevant to swapping are surprisingly lax. The important issue here is that the kernel be able to accurately determine where objects are located at any given instant (or at least have a very good idea; it is rather difficult to state precisely where a given segment is while it is in the process of being moved in or out of memory).

Interestingly enough, the two swap calls are likely candidates for expulsion from the kernel. If the kernel were willing to allow a process to label (name) core segments, then by adding a single "Name-core-segment" kernel call one could eliminate both Swap-in and Swap-out kernel calls, and at the same time remove some of the

---

[1]. In the current implementation, processes are invoked by name rather than process number, but modifying the code to use numbers instead would represent a trivial change. If one allows the initiator, for instance, or even those processes which are allowed to create other processes to communicate with the CPU scheduler via the Send-Message kernel call, then one is able to reclaim any priority scheduling flexibility one might have lost by merely sending process numbers to the CPU scheduler instead of names. The creating process presumably knows the name of the process it has created and could be returned a process number by the kernel. The creating process could then communicate a priority to be associated with that particular process number by sending a process number, process priority message to the CPU scheduler.

strangeness inherent in the kernel's handling of disk completion interrupts. [2]

There are several other security relevant actions which impact when swapping can occur, however. A communications path between processes in the system is readily seen if one allows processes to modify segments which are in the process of being swapped out. [3] Likewise, serious consequences can result from swapping out a segment which is involved in some on-going I/O operation. Therefore attention needs to be given in the Swap-Out call that 1) the segment is indeed "swappable" -- that is, no asynchronous access is being made to it -- and, 2) no further access will be made to it until it has been swapped out.

Assurance 1 can be given if one checks at swap time whether any device I/O has been initiated which will access the memory segment involved in the swap. If a device is

---

[2]. When an I/O interrupt on a disk occurs, the kernel must check to see if the request just completed is a swap request and perform a number of additional operations if it is.

[3]. In particular, if a process is allowed to modify segments while they are in the process of being swapped, it is now able to sense when swapping has occurred. Hence it has in effect been given information by the CPU scheduler and the CPU scheduler can nc longer be classified as a pure information sink. See [POPEK74A] for a further discussion of the notions of informatica sources and sinks.

performing an asynchronous operation on a memory segment, that segment will have been "locked" at the time the kernel Start-I/O call on the device was executed. If the segment is still locked at swap time, then the swap cannot be allowed to proceed until the segment is unlocked by the eventual completion of all I/O operations which caused it to become locked. Once the swap is allowed to proceed, further asynchronous modifications to the memory segment can be prevented by marking it as "not in core" for all operations except swapping. [4]

Assurance 2 cannot be given merely by marking the segment as not in core, since this would cause the eventual Start-Indirect-I/O call made by the disk scheduler to fail. Accordingly, some special access must be used in the eventual Start-I/O for the swap-out request, and all process relocation registers which point to the segment modified such that any attempted access will fail.

Hence, if one pushed the two swap calls into the normal Start-I/O call, the special checks already necessary for swaps would change slightly, becoming instead an "is the segment locked" check rather than one of "is this a

---

[4]. Special checks must be made at the time of the actual Start-I/O call which causes the swap to occur, otherwise the fact that the segment does not "appear" to be in memory would prevent the segment from being swapped out.

swap request?". The only operations saved would be the kernel construction of the "correct" swap request, forcing this task onto the CPU scheduler, instead, and perhaps giving the CPU scheduler a finer grain of control over swapping. [5] In addition, one has probably given the CPU scheduler the extra "privilege" of labelling memory segments. [6]

---

[5]. The qualifier "perhaps" is added here because the level of control the CPU scheduler currently has over swapping is not at all clear. Such control depends to a large extent on issues such as naming conventions which have not yet been thought out in detail.

[6]. Here one becomes a bit squeamish, for this is clearly data security relevant. One could add extra constraints, however, which would limit what names the CPU scheduler could associate with the memory segment. Alternatively, one could push the name change back into the interrupt handler and eliminate the new "change names" kernel call. The total result then would be two less kernel calls (Swap-In and Swap-Out) at the expense of some additional complications to the Start-I/O call.

5.4 Other Kernel Primitives: Design

5.4.1 Attach-Segment Design

For efficiency reasons, the mechanism for attaching segments was implemented using the hardware memory management facilities. The only other alternative would have been to support segment attachment, accessing, and detachment in a purely interpretive manner -- an alternative clearly unreasonable from a performance and efficiency standpoint. Since segment accessing capabilities are based upon the facilities provided by the memory management hardware, the available access rights to segments are limited to those supported by the hardware -- namely, no access, read access, and read/write access. Two additional access types become available when one adds execute access -- read/execute access and read/write/execute access. These last two access types can be associated only with segments attached to I-space relocation registers, however, since instructions are fetched by the CPU using the I-space relocation registers.

In order to gain the capability for efficient accessing of core-resident code and data, a process executes an Attach-Segment kernel call. The Attach-Segment kernel call performs security relevant operations by appropriate loading of the process relocation registers in

such a way as to allow certain subsets of the possible accesses to proceed without further kernel intervention. Proper loading of the process relocation registers is, of course, essential from a data security viewpoint, although certain classes of errors would not fall directly under the heading of data security flaws. [7]

In addition tc the restrictions imposed by the hardware on the valid access types which may be associated with a given segment, the kernel has placed additonal restrictions upon the use of several of the supervisor relocation registers: several are reserved for the kernel/process communications buffer, are initialized by the kernel to be mapped into the communications buffer, and cannot be released or reattached by the process. This last restriction is in some sense unnecessary as long as the kernel does not place information in the communications buffer using the process relocation registers. The contents of process relocation registers and whether they are currently attached is of little interest to the kernel as long as 1) the relocation registers reference segments which are legally accessible by the process, and, 2) the

---

[7]. If the kernel "accidently" attached segment X to relocation register R with read/write access when in fact it was asked to attach segment X to relocation register R with read access, no security violation would have occurred -- provided that the process possessed the potential for attaching segment X with read/write access.

kernel does not use the process relocation registers for accessing the communications buffer. [8] Thus the restriction that the process cannot detach the communications buffer relocation registers is one which might easily be relaxed. However, since the communications buffer is not a real segment, once its associated relocation registers have been released the process has no means of reattaching them.

The other point of interest in the Attach-Segment call is the notion of indirectly attaching a segment to another process. This ability is necessary for the initiator in order to establish enough of the process's necessary environment for it to bootstrap itself into core. Once the process's bootstrapping code is in memory and properly attached, the process can, of course, be held responsible for bringing its other components into core.

A number of alternative methods for initializing

---

[8]. If a process is allowed to detach or reattach the supervisor relocation register which references the communications buffer, then under certain circumstances it is conceivable that a kernel memory management trap could occur. If the kernel were to reference the communications buffer via MTPI, MTPD, MFPI, or MFPD instructions and if the segment was either not attached or was attached with inappropriate access, then a kernel memory management trap would occur. It is, of course, unnecessary for the kernel to reference the communications buffer in such a fashion since all process communications buffers are mapped into kernel space as well as process space. It might, however, be more efficient to access the communications buffer of the last running process in such a manner.

process memory exist but will not be discussed here since they have not been examined in any great detail. Each alternative method in the end still rests on the notion of initially bringing a minimal amount of code into memory which then "knows" enough concerning process requirements to bring in the next piece of process code which then knows enough to bring in the next incremental chunk, etc. The task of attaching segments is crucial in this bootstrapping mechanism, and although a certain restrictive bootstrapping mechanism could be built into the Create-Process call itself [9], the indirect attach mechanism is of much greater generality and is available at little extra expense.

As currently designed, the control one has over invocation of indirect segment attachment may not be as flexible as might be desired. A process which is allowed to indirectly attach segments to another process is allowed to attach any segments which the given process itself could have attached directly. Currently there is no way of limiting this indirect attach capability to particular

---

[9]. One could, for instance, place standard process bootstrapping code into an executable portion of the communications buffer at process creation time. Since the communications buffer is automatically attached to the new process by the kernel, it would be relatively simple to require that the process begin its execution at the bootstrapping code so placed. This mechanism would be quite limited however, since it is desirable to keep the communications buffer as small as possible.

segments: process X cannot have indirect attach segment access to process Y for segment Z without also possessing indirect attach segment access to process Y for segment W, given that Y can attach both W and Z. Thus the indirect attach segment privilege is associated on a process basis rather than on an individual segment and process basis. Although this latter ability may be desirable it involves complications to the structure of the kernel's underlying protection data structure and so has been excluded from the design.

5.4.2 Release-Segment Design

The issue of interest regarding the design of the Release-Segment kernel call concerns the circumstances under which a segment can be released. This issue, however, breaks down into several inter-related components: 1) what segments can be released, 2) when a segment can or cannot be released, and, 3) who can cause a segment to be released.

Generally speaking, any segment attached to a given process can be released by that process, with the exception of those portions of the supervisor portion address space reserved for the kernel/process communications buffer. [10] Additionally, any process X possessing the "indirect release segment" capability for process Y may indirectly release any segment attached to process Y which process Y itself could have released. This indirect release facility is not strictly necessary, but it is available with virtually no extra complications to the kernel code and is particularly convenient for the initiator. [11]

---

[10]. Under certain circumstances it is conceivable that, in principle, even this reserved area could be detached. This particular issue is addressed in the section on Attach-Segment design.

[11]. This indirect release capability would be absolutely necessary should the internal arrangement of the Destroy-Process kernel call change in such a way as to require all segments be detached before the process can be destroyed. Such a change is likely since it decreases the amount of code required in the Destroy-Process call.

5.4.3 Create-Segment Design

The most important design questions relative to the Create-Segment kernel call arise with respect to 1) how the segment is allocated, 2) when the segment is to be zeroed, and, 3) how the system object directories are updated.

In order to destroy the residue information resident in segments which have been deallocated, segments must be zeroed before they can be reallocated. There are two obvious times when this zeroing operation can be conveniently done: 1) when the segment is created (allocated or reallocated), and, 2) when the segment is destroyed (deallocated).

On the surface, both alternatives seem equivalent. If all segments are zeroed the first time the system is ever brought up, and afterwards either zeroed at creation or deletion time, then it would seem to be impossible for one process to steal residue information from another process by examining the contents of newly allocated segments. Under ideal situations this is true. Under slightly less than ideal circumstances, it is not.

As long as one assumes that the system never fails -- that system crashes, hardware or software induced, never occur -- the two alternatives are effectively equivalent. However, once one moves into a less idealistic environment

where periodic system crashes may be an unwelcome fact of life, the total equivalence of the two alternatives vanishes. The non-equivalence is easily seen in the case where one zeroes the segment at destruction time and the system crashes while in the process of destroying the segment, after altering the directory but before the zeroing operation is completed. The segment then appears to be free when in fact it is not. Reallocation of the segment transmits its previous, non-zeroed contents to the new owner, and a security leak has occurred.

Three logical operations must occur before the creation or destruction of the segment can be considered completed: 1) the disk space for the segment must be allocated (deallocated), 2) all process access to the segment must be added (removed), and 3) the segment must be zeroed. Each of these operations requires some finite amount of time. Although the kernel operates in a non-interruptible fashion, it seems fairly obvious that it cannot afford to simply hang waiting for all required actions to complete. Hence it is meaningful to discuss the sequencing of these operations in a manner which assumes arbitrary interruptibility between them.

Breaking these three steps down for both allocation and deletion of segments, allocation requires 1A) disk space allocation, 2A) capability addition, and, 3A) segment

zeroing. Deletion requires 1B) disk space deallocation (freeing), 2B) capability revocation, and, 3B) segment zeroing.

Examining the allocation operation first, clearly 3A must occur before 2A or the passage of residue information that segment zeroing was designed to prevent can occur if the system crashes after 2A was completed but before operation 3A had finished. It is just as obvious that 1A must occur before 2A or one may have extended a capability to an object that does not physically exist. Similarly 1A must occur before 3A for the simple reason that it is difficult to zero a segment whose location is not yet known. Hence the correct ordering among the operations necessary during segment creation should be: A) segment disk space allocation, followed by B) segment zeroing, followed by C) addition of capabilities to the segment to the protection data for the new owner of the segment.

For segment destruction the sequence is slightly different. If one wishes to provide a job restart capability over system crashes, then 3B cannot occur before 2B -- otherwise the original information contained in the segment which was destroyed is lost forever, possibly along with files created by the job. Likewise, 1B cannot be done before 3B or residue information may be transmitted to the new owner. Thus the proper ordering with respect to

segment destruction should be: A) revoke all access to the segment being destroyed, B) zero the segment, and, C) free the disk space associated with the segment.

Thus the sequencing of operations is slightly different in the two cases. Which of the two alternatives is safer (or has less disadvantages)?

If one considers the consequences of a system crash between any two steps of either sequence, then a crash between 2B and 3 results at worst in a non-zeroed partially freed segment -- but no security violation since the segment is only partially free. A crash between 3B and 1B results in the complete loss of a segment, but violates no data security constraints. [12] If one zeroes the segment at creation time, then a crash between 1A and 3A results in a partially allocated, non-zeroed segment, but no data security flaw results since the segment is only partially allocated. A crash between 3A and 2A results in a non-free, zeroed segment.

Both alternatives have nearly equivalent results, but in the case of zeroing the segment at creation time, the security necessary actions are not separated in time from the actions which necessitated those actions. This alternative -- that of zeroing segments at creation time

---

[12]. A viability question arises here, however.

rather than at destruction time -- was the alternative chosen to be implemented by the security kernel in the Create-Segment kernel call.

5.4.4 Destroy-Segment Design

The most interesting design decision relative to the Destroy-Segment kernel call was the decision to zero segments at creation time rather than at destruction time. A discussion of this issue can be found in the section concerning the Create-Segment kernel call design. One additional point is of interest here: under what conditions can a segment be destroyed? Since some policy decisions have serious security implications, it is appropriate that this question be addressed here.

Since the decision had already been made to zero the segment when it is created rather than when it is destroyed, it is important that all possible access capabilities relative to the segment be revoked when the segment is destroyed. This revocation of capabilities is not merely a matter of deleting entries from the protection policy database -- it is not merely a matter of arranging things so that EVAL no longer answers "yes" when questioned concerning the segment. One must also remove all "dynamic" accessing capabilities to the segment.

In the UCLA system, these dynamic capabilities are inferred from the attached/unattached status of the segment. If the segment to be destroyed is not attached to any process when the Destroy-Segment call is attempted then

the operation proceeds normally, as expected. If, however, the segment being destroyed is attached to some process, then basically two choices exist: 1) the Destroy-Segment call can be made to fail and continue to fail until such time as the segment is eventually released from all processes having it attached, or, 2) the Destroy-Segment call can automatically detach the segment from all processes having it attached and then destroy the segment. From a strict data security standpoint, the two alternatives appear equivalent. The notion of least common mechanism, however, suggests that the first alternative is more desirable since it involves fewer kernel actions than the second. [13]

More importantly, however, alternative 2 imposes a scheduling policy on the system which is not strictly necessary. In choosing the second alternative, one might find the grain of control over when segments may be indirectly released insufficient: if process A possesses the ability to destroy segment B, then it would also implicitly possess the ability to indirectly release segment B from any process C having it attached -- indirectly release that segment from any process having it

---

[13]. The code for actually detaching segments is needed by the Release-Segment kernel call already, however. Thus the "extra" code required by the second alternative need not be considerable.

attached -- independent of whether process A explicitly possesses the ability to indirectly release segments from process C. Alternative 2 thus imposes a priority structure upon the utilization of segments. If implemented it would give any process possessing destroy-segment access to a given segment preemptive power over any other processes attempting to use the segment -- regardless of the relative importance of those other processes. This did not seem a wise (or even useful) policy to impose upon the system, especially since alternative 1 shows it to be an unnecessary imposition. Should it prove desirable to implement such a preemptive, priority structure, the indirect release-segment ability supported by the Release-Segment kernel call, combined with the Destroy-Segment call should provide a sufficient base from which to reconstruct the features offered by the second alternative. Accordingly, the first alternative was chosen as the one to be implemented by the kernel.

Additionally the requirement was added that the segment must be currently unlocked before it can be destroyed in order to avoid the complexities involved in any attempt to halt an I/O operation in mid-stream.

5.4.5 Sleep Design

The Sleep kernel call involved fewer design decisions than any other kernel call. As designed it contains no security checks whatsoever. It merely causes a immediately context swap between the process performing the call and the CPU scheduler. It was included in the kernel strictly for reasons of system viability: since processeses are prohibited from executing actual WAIT instructions by the hardware [14] , another method had to be devised if one wishes to reclaim CPU time which would otherwise be wasted in some sort of "wait until there is something to do" loop.

Due to the unusual nature of the pseudo-interrupt mechanism which was developed and discussed earlier, the pertinent features of the Sleep call cannot be synthesized through a combination of a Send-Message call informing the CPU scheduler that the (sending) process does not wish to run, followed by an Invoke-Process call to invoke the CPU scheduler. Such a combination can fall victim to the lost interrupt problem since the process wishing to sleep is interruptible between the time it attempts the Send-Message

---

[14]. The ability to execute machine WAIT instructions would not provide the process with any additional power, however, since the kernel fields all interrupts itself.

the time it attempts the Invoke-Process call. [15]

The Sleep call is, however, functionally equivalent to the Invoke-Process call except for added information which is transmitted to the CPU scheduler. If the Invoke-Process call were to be modified to inform the process being invoked of the process identity of its invoker, however, then the Sleep call could be synthesized by a combination of a Send-Message from the process to the CPU scheduler indicating a desire to sleep, followed by an Invoke-Process to the CPU scheduler, combined with an "understanding" between the CPU scheduler and processes which invoke it that "sleep" requests will be ignored unless followed reasonably quickly by a successful invoke by the process wishing to sleep. [16]

---

[15]. If an interrupt comes in for the process requesting the sleep after it has sent the message to the CPU scheduler, but before it can invoke the CPU scheduler, and if the CPU scheduler has been invoked by a clock interrupt in the meantime, it is possible that the CPU scheduler will still think the process wishes to sleep when in actuality the awaited for interrupt has already occurred.

[16]. How one can define "quickly" in such contexts is not at all clear. Presumably the CPU scheduler would be able to distinguish between being invoked because of a timer interrupt as opposed to being invoked by a particular process. The CPU scheduler could perhaps assume that if the sleeping process did not invoke it between two successive timer interrupts that the sleep request should be ignored. Although this might cause a sleep request to be ignored which should not have been, it does not result in lost interrupts.

5.4.6 Send-Message Design

The Send-Message kernel call was motivated by two precise needs: 1) in order to make capability faulting a reasonable mechanism, the process that received the capability fault had to be able to cause the proper kernel protection data segment to be brought into memory, which required that the process be able to communicate the name of the necessary segment to the CPU scheduler; and, 2) in order to make limited data sharing between processes via shared read-write segments, some limited communication facility was necessary in order that the processes involved agree as to the name of the segment to be shared. [17]

Both cases represent situations where a small amount of information needs to be passed from one process to another. In the first instance, the amount of information passed need only be as large as that necessary to hold some representation of a segment name, plus some flag to indicate to the CPU scheduler that this message concerns a

---

[17]. This could have been avoided by placing some rather severe restrictions upon how shared segments are to be used. For example, one could require that whenever process A and process B wish to communicate they do so via shared segment A-B, that process B and C communicate via segment B-C, etc. Such a solution does not generalize well, however, to situations where more than two processes wish to communicate with each other, nor does it handle the question of how process A becomes aware that process B wishes to communicate with it.

to swap in a Kernel protection data segment. [18] Is such a limited facility sufficient to handle case 2?

All that is really required is that the kernel act in a manner similar to that of an answering service: Process X calls the kernel and asks to be connected to process Y over line Z(where Z is, for instance, the name of a shared segment X wishes to use in connection with its conversation with Y). The kernel then notifies process Y that it has received a call from process X and that the return call should be placed over line Z. Now Y knows that X wishes to use shared segment Z as a communications path between them. Both processes have thus been effectively connected one to the other over line Z. [19] Any other protocols needed for communication are left to the imagination and discretion of the processes involved. Having set up the initial dialogue, the kernel is no longer concerned with what is actually said.

What security issues are involved here? At first glance it appears The only question which has any security

---

[18]. This is assuming, of course, that the CPU scheduler is never sent any other type of message. If this is not the case, then this message type flag is unnecessary.

[19]. This is quite similar to the Initial Connection Protocol (ICP) in current use on the ARPA network. The only major difference is that the network callee specifies the new number to be used, rather than the caller, as is the case here.

relevance is whether to allow the two processes to communicate in the first place. Once the decision is made to allow the message to be sent, the actual content of the message is of no interest to the kernel whatsoever.

Perhaps more at stake here than security considerations are certain viability questions. Suppose one process is a file management process and that a large number of other processes normally wish to communicate with it. It seems most reasonable to expect that the file management process might be designed such that it is prepared to accept messages from anyone. Now the question arises as to where these messages are going to be placed. If they appear in the queue of the receiver, then the receiver may find it extremely difficult to insure that its queue does not overflow, especially if it is very lenient concerning with whom it is willing to communicate.

If it is more discriminating (would accept messages from only one or two specific processes, for instance) and if some mechanism existed whereby it could prevent processes from sending more than one message at a time, then the queue overflow problem might be effectively dealt with provided that the queue itself is physically large enough to hold the combination of the number of outstanding I/O request notifications, pending interrupts, and the requisite number of messages. However, clearly the queue

will not be large enough in the general case.

What can be done then? The desires of this hypothetical file management process seem most reasonable. It would not be too difficult to devise some sort of locking mechanism with regard to messages which would prevent a given process from sending multiple messages to a given destination -- effectively a one-at-a-time capability which is revoked once exercised and reinstated only at the request of the receiver, for instance. This, however, still does not solve the queue overflow problem of our hypothetical file management process. There still is not enough queue space to avoid overflow in all instances.

Several alternatives came to mind: 1) do not put the message in the queue, 2) limit the number of messages a given caller can have outstanding, store the message with the sender, and provide a kernel call to move the message to the receiver's space when so requested, 3) store the message with the receiver and limit the number of messages it can receive at any given time, effectively refusing to send messages to him when his allotted message space is full.

Alternative 2 requires yet another kernel call, and hence is probably not reasonable unless all else fails. Alternative 1 is embodied in alternative 3 (and in

alternative 2 as well), thus both can be implemented by the third alternative if it contains no serious problems of its own.

The major drawback possessed by the third alternative is that it requires additional space be reserved in the kernel for these message slots since the logical place to put them is in the shared kernel/process communications segment. [20] This need not be a serious handicap, however, if some small upper bound can be placed upon the size and number of these message slots. If the restriction is imposed that a maximum of one message can be sent from process A to process B at any one time, then it is relatively easy to place an upper bound on the maximum number of necessary slots: simply N*N, where N is the maximum number of processes allowed to be active at any given moment (or N*(N-1), if a process is not allowed to send messages to itself). If one then chooses the maximum size of a message slot to be relatively small, then the space problem virtually disappears.

Placing the message slots in the communications

---

[20]. These message buffers must at least remain core-resident at all times since messages, like interrupts, occur asynchronously.

segment eliminates the need for an additional kernel call in order to read the messages since the process can now read them directly. The "one-at-a-time" property of the messages can be easily implemented by placing a full/empty flag for each message slot in a process-writeable portion of the communications segment, which the kernel interrogates before placing a new message in the slot. If the message slot is full, then the kernel simply refuses to copy the message from the sender to the receiver. This provides a double level of protection for the receiver since it can now inhibit messages from processes which possess the potential of communicating with it. [21]

---

[21]. Effectively there are two ways a Send-Message call can fail: 1) because the sender can never send messages to the intended receiver; and 2) because the intended receiver is not currently prepared to receive messages from the sender. This will probably prove to be of great utility with regard to processes such as the initiator or the file management process which are potentially willing to listen to anyone.

5.5 Kernel I/O Primitives: Design

5.5.1 Attach-I/O-Device Design

The Attach-I/O-Device kernel call had three primary motivations: 1) a mechanism for allocating I/O devices to user processes was necessary; 2) some mechanism was needed to "pass" consoles from their initial owner, the initiator, to their proper owner, the VM to which they should belong, and 3) some mechanism was necessary to provide for the returning of consoles to the initiator when the user wishes to halt his current VM or attach himself to another process.

These needs could have been accomplished through two kernel calls -- one to attach an I/O device to one's own process and another to attach a device currently attached to one's own process to another process. It was, however, fairly easy to collapse the two calls into a single kernel call by supplying an additional parameter: the name of the process to which the device is to become attached.

Once the decision to merge the two possible attach calls into a single attach call had been made, the question of how a process "discovers" it has been given a console arose. This issue was present already if an "indirect attach console " call were to be implemented. The most

obvious solution might have been to send a dynamic, asynchronous (to the receiver) message to the new "owner", telling him of his acquired device. Such a solution, however, required finding a place to put the message. The wrap-around queue used for sending interrupt notification seemed a likely candidate, but if a process can receive an arbitrary number of "attached console" notifications, it becomes exceeding difficult for the process to insure that its queue does not overflow. The process can no longer estimate the maximum number of entries which may be thrown into the queue at any point in time. This solution would also make it possible for other malicious processes to drastically affect the performance of a given process by causing its queue to overflow, if one were to extend the generalized "indirect-attach" capability to processes other than the initiator. (User processes, of course, need to be able to attach consoles to the initiator.)

Other obvious alternatives were no more attractive: 1) another message passing area could be allocated for the process of a fixed size with some sort of flag to indicate whether it is full or not, or 2) the kernel could manage a number of "message" queues in its own space and provide a mechanism whereby a process can read its messages.

The first alternative is unsuitable in the case of the initiator, unless the size of this message passing area is

large enough to hold a number of messages equal to the maximum number of consoles in the system. Otherwise it is possible for the initiator to "miss" notification of a reattached console.

The second alternative requires additional queue management in the kernel, complete with overflow problems, and also requires an additional kernel call to let the process read its messages.

The mechanism finally devised is a derivative of the additional message area scheme, but in its most simplistic form. Enough space is allocated in the shared kernel/process segment for every device in the system to have a slot. The key here is that each "slot" consists of only a single bit, indicating whether the device is currently attached or not. This makes it relatively simple for the kernel to notify a process that a device has been attached or detached -- no clumsy links or pointers to follow down, no queue management or overflow problems to bother with. [22]

It is also rather simple and straight-forward for the process to notice that a console has become attached: it simply keeps its own record of which devices it thinks are currently attached and periodically compares its own list with the kernel's list of attached devices and updates its

own list accordingly. The process no longer needs to worry about its queue overflowing simply because some other process has passed it a console.

---

[22]. This "attached device" notification could be conveniently sent to the new owner by the process causing the device to be attached via SEND-MESSAGE, if the two processes are allowed to communicate. However, removing the notification from the attach call implies that the correct functioning of any process which has devices attached to it by other processes may now be dependent upon the correct functioning of other processes. This is an important consideration, especially with regard to the initiator which will typically have devices reattached to it and which needs to be eventually verified. For this reason, the decision was made to retain the the notification in the attach call.

5.5.2 Release-I/O-Device Design

The kernel has not particularly concerned itself with any of the gory details of device allocation -- not even for dedicated devices. If a process is allowed to attach a given device and that device is currently unattached (free), then the kernel simply allocates the device to the process which requested it. [23]

It would be unrealistic, however, to assume that all processes behave correctly all of the time, and it is quite conceivable that under certain circumstances it might become desirable to be able to detach a given device from its current owner without necessitating the restarting of the entire system. [24]

How might this be done? Two immediate alternatives are

---

[23]. One could build a more sophisticated device allocation scheme outside of the kernel, however, by creating a device allocation process which would initially "own" all dedicated devices not belonging to the initiator. This device allocation process could then "pass" devices around in a fashion similar to the initiator.

[24]. Such a desire is not at all unreasonable. One can easily imagine a situation where a malfunctioning virtual machine has managed to "tie up" the user console so effectively that the user is unable to communicate with the virtual machine monitor long enough to halt the virtual machine. In such instances, one would like to be able to walk up to another console and by conversing with the initiator eventually salvage the user's original console and job.

obvious: 1) the Release-Device kernel call could take as an extra argument the name of the process from which the device is to be released, and special processes (the initiator, for instance) given an "Indirect Release Device" capability; or 2) a detach device facility could be implicitly included in some other kernel call (Stop-Process, for instance).

Just what is involved in releasing an I/O device? Nothing at all really, if the device is inactive at the time. If the device is currently active, however, it is an entirely different story, since attempting to stop an I/O operation in mid-stream often causes unpredictable results. Thus, one would much rather have the device stop of its own accord before attempting to release it.

But how is it possible to insure that the device will eventually stop? How can one insure that the owner of the device does not somehow "sneak in " while the kernel is waiting for the I/O to complete and manage to start another I/O operation on the device, so that the device never really quiesces?

Let us stop for a moment and consider the problem logically. The fact that a Release-Device call for the device was made and found the device busy implies that the process to which the device was attached was not running at

the time (unless, of course, the process which requested the release has the device attached to itself). Now, if one could insure that the process to which the device is attached will not be allowed to run again, then it would be impossible for any additional I/O to be initiated on the runaway device.

There is a relatively straight-forward way of insuring this: One merely arranges matters so that the CPU scheduler never invokes the process again until after the device has finally been recovered. There are essentially two ways of doing this: 1) one can depend upon the "good will" of the CPU scheduler and hope that if one asks the CPU scheduler not to invoke the process that the scheduler will comply with that request; or 2) one can insure that the CPU scheduler can not invoke the process, even if it wishes to do so, by revoking the scheduler's "Invoke Process" access to the process in question, at least temporarily.

Two kernel design principles are in conflict here. The principle of minimization of maximum damage suggests that the second alternative is the safer of the two. [25]

---

[25]. One cannot, however, keep the CPU scheduler itself from running since it will periodically be invoked by the kernel itself. On the other hand, it is unlikely that one would ever have occasion to grab a device away from the CPU scheduler anyway, since it is rather unlikely that the CPU scheduler would ever have an attached device in the first place.

This latter function, that of making a given process uninvocable, is precisely the function performed by the Stop-Process kernel call.

On the other hand, the issue being addressed here is strictly one of viability and not one of data security. The worst that could happen here is that a malevolent or malfunctioning CPU scheduler could deliberately continue to invoke a process which the initiator (for instance) is trying desperately recover a device from or destroy. The maximum damage to the system is that a process might be able to prevent itself from being destroyed.

Thus the principle of least common mechanism (the "keep the kernel small" principle) suggest that the function performed by the Stop-Process kernel call can be safely excised from the kernel if no other data security induced motivations for the Stop-Process call can be advanced. [26]

---

[26]. Indeed, it seems quite likely that the Stop-Process call may be removed. The decision is not yet final, however.

5.5.3 Start-I/O Design

One of the most obvious security relevant aspects of almost any operating system is its I/O mechanism. If one wishes to demonstrate that it is impossible for unauthorized users to gain access to the contents of an important file, it is clearly necessary to demonstrate that it is impossible for that file to be read by unauthorized users. If the machine's basic I/O hardware is such that this restriction cannot be enforced in the basic hardware, it becomes necessary to provide the requisite protection in the system software. Since virtually no protection is provided by the I/O hardware on the 11/45, our kernel is given the task of ensuring that no I/O initiated by any user process can compromise the security of any information of another user without specific authorization being indicated in the security policy of the system.

Due to the nature of the 11/45's I/O structure and its lack of specific I/O instructions, the kernel usurps all I/O capabilities from supervisor and user processes [27] by eliminating access to the uppermost 4K of the machine's real memory for user and supervisor processes through use of the memory management facilities of the machine. In

---

[27]. On other machines this usually happens "automatically" since I/O related instructions are usually privileged.

doing so we have taken from the user his ability to perform I/O directly. A program which is unable to do I/O, however, is severely restricted in its usefulness. Accordingly, we were forced to substitute an alternate mechanism to replace the original I/O facilities now usurped by the kernel.

In removing the usual I/O mechanisms, we replaced them with several system calls (kernel calls, analogous to supervisor calls or SVC's in other systems). These kernel calls initiate I/O, request the initiation of I/O, and return status information to the process performing the call. In many cases the interface provided by the kernel Start I/O call closely resembles that interface existing on the real machine. In other cases, however, the interface is much simpler than the interface supplied by the bare hardware.

The kernel contains a set of "device-driver-like" routines for each device type. These device-dependent routines are used in determining the validity of I/O requests for the corresponding devices as well as in the loading of the individual device registers necessary for the initiation of the I/O itself. Although each new device type requires a set of kernel routines, many of the required actions are common for all devices and a significant amount of code sharing among them is possible.

For instance, the UCLA IMP hardware interface expects to be supplied with a starting pointer and an ending pointer from which it determines where to get or put its data, as well as the number of bytes to be transfered, with all sorts of 'funny' conditions occuring when the starting or ending address is odd. The interface to the IMP provided by the kernel is much simpler, although the complexities of programming the real hardware interface still remains in the kernel itself: the user process specifies a combination of segment name, segment byte offset, along with a byte count for the number of bytes to be transferred and the kernel then transforms them into the form required by the actual hardware.

One might argue that it is unnecessary for the kernel to perform this translation, that indeed it would reduce the complexity of the kernel code and hence ease its proof if the user process were required to do the translation and the kernel thereby avoid it. This is a reasonable suggestion which might have been implemented were it not for the simple fact that the kernel, for security reasons, is required to verify that the real addresses to be used by the I/O device fall within the address space of the process doing the I/O and will not trespass into areas belonging to other processes.

Also, some address translation is required anyway unless the process is allowed to know where in real memory its virtual addresses reside. Such knowledge would result in additional complexity when swapping is introduced into the system, as well as pushing the added burden of keeping the user processes informed of their real addresses onto the kernel. Rather than do so, we chose to perform a single level of virtual to real address mapping inside the kernel, a process which is greatly simplified by couching process virtual addresses in standard forms rather than in completely virtual, device dependent ones. [28]

Most I/O operations have several features in common, regardless of the physical I/O device involved. They operate upon some location in memory, transfer some specified number of bytes, either cause or fail to cause an interrupt to be generated, and perform some well-defined operation on the memory locations involved.

Accordingly, the standard arguments to the kernel

---

[28]. We could have allowed the process to specify its virtual addresses in the appropriate device-dependent format and then converted those virtual addresses to real ones in a device-dependent fashion. This is a somewhat messy operation in general since device register formats are non-standard. The complications of performing 18-bit addressing on a 16-bit machine enter here in full force. Here the extra 2 bits of the address are positioned within the device registers in such a way as to make even "simple" double-word arithmetic all but impossible.

Start-I/O call provide these pieces of information. The starting memory location is identified by a combination of segment name and segment byte offset (from the beginning of the named segment). The number of bytes to be transferred during the operation is given as a byte count, whether a pseudo-interrupt is to be generated is specified by an interrupt flag, and the operation to be performed upon the memory locations is given by an operation type.

5.5.4 Status-I/O Design

The Status-I/O kernel call involves few essential design decisions. One of the major questions, however, was whether the call itself was indeed necessary. Since interrupt notification entries contain device status information, a separate kernel call to provide device status information is in some sense redundant.

However, since the original bare machine I/O itself is capable of being utilized in both polling and interrupt driven modes, it seemed reasonable to provide equivalent facilities. Additionally, if one required all I/O to be performed in interrupt mode only (from the viewpoint of the virtual machine monitor) without providing any facility for device status interrogation, this places a perhaps unreasonable constraint on the virtual machine operation. If a VMM has no means of determining real device status while the I/O is active, it has no way of reflecting status changes back to the virtual machine until the I/O has completed. The virtual machine, however, might have been designed in such a way that it might "time out" certain I/O operations if they did not complete within a certain time interval. [29] If the VMM itself has no way of obtaining status information on devices which have not yet completed, then reflecting the information which results in such time-out activities to the VM becomes more complicated.

The actual code for the Status-I/O kernel call, however, did not represent new code to be added to the kernel to any great extent. The code to return the required information to the user was already necessary for the kernel to return interrupt notification.

The only moderately interesting design decision relative to the Status-I/O call was the decision not to allow processes not in direct possession of a shared I/O device to do status calls on the device. The consensus was that little use could be made of such a capability and that the consequential complication in the kernel is unnecessary.

---

[29]. Such is the case with certain network type protocols. Since one of the most interesting virtual machines we are interested in running communicates with the ARPA network via the established protocols, it is important to at least consider what is required here in order to allow the virtual machine monitor to be able to virtualize such time-out behavior.

5.6 Kernel Shared I/O Primitives: Design

5.6.1 Request-I/O Design

The Request-I/O kernel call was directly motivated by the shared I/O mechanism. It is the function of a shared device scheduler to accept I/O requests from other processes wishing to utilize its device, to decide when those requests should be allowed to proceed, and to actually cause the final initiation of the process requested I/O. In order for any of this to work, however, it was first necessary to devise some means of "sending" the request to the appropriate device scheduler in the first place. This was not so simple a task: Thus far no communication mechanism between processes existed, and a large amount of thought had to be given as to the "best" way of handling the problem.

Furthermore, unless one is willing to take the correctness of any such device schedulers on faith, this must be a rather peculiar sort of message being sent --one that can be read, but not written, and yet one that is easily distinguishable by the kernel as a process-requested I/O request. It is necessary for the kernel to be able to determine that the I/O to be done by the device scheduler on the process's behalf, is indeed the I/O which the process itself originally requested, or that, for that

matter the process ever made such a request in the first place. If this were not the case, it would be possible for a malevolent device scheduler to arbitrarily initiate I/O, ostensibly on another process's behalf, which was never even requested by him, thus subverting the process's data and invalidating any process isolation guarantees which might have been claimed.

What is needed, then, is a message which can be read by the device scheduler, which can be neither faked nor overwritten, and can be earmarked by the kernel as a "legal" indirect I/O request. (But not necessarily a valid one -- the original requestor may be trying something it is not allowed to do.)

The concept of locked-boxes was devised to meet the constraints mentioned above. When a process executes a Request-I/O kernel call, as an argument to the call it gives a complete I/O request in a format identical to that required by the Start-I/O call. The I/O request is not itself checked for validity at the time of the execution of the Request-I/O call, but is merely bundled up and placed somewhere where it cannot be modified after first prefixing it with the process name of the requesting process. The kernel "remembers" where it has put this request and in the locked-box itself is some indication of whether the locked-box currently contains a request. Some indication

of the location of this locked box (its name, an index, etc.) is then sent to the owner of the shared device. When the scheduler starts the request via Start-Indirect-I/O, it specifies the request to be started by naming the locked box. If the box so named contains a request for the device owned by the scheduler, then the start is performed and the box freed so that a second start attempt on the part of the scheduler will fail.

Two major questions arise in connection with shared I/O requests and device scheduling: 1) how does the device scheduler gain access to the information it needs to schedule the device; and 2) where should these locked-boxes be kept.

The first question, then, is how the scheduler for the shared device is provided with enough information about the request to do some sort of reasonable scheduling on the request other than a simple first-come-first-served mechanism. [30]

---

[30]. If first-come-first-serve is the only type of scheduling possible based on the information available to the scheduler, then it is probably more reasonable to eliminate the device scheduler (and the extra overhead it involves) entirely and force processes which wish to use a shared device to compete for it, just as if it were a normal, dedicated device. One would probably wish to enforce that the device be released automatically upon completion, however, in order to prevent a single process from tying up a much-needed resource indefinitely.

As currently specified in the Request-I/O call, however, the locked-box containing the request contains only logical device addresses-- that is, segment names and offsets -- which must somehow be mapped into physical device addresses in order for any reasonable device-dependent scheduling to be done. It is probably unreasonable to assume that every device scheduler has enough accurate, up-to-date information concerning all possible segments on its device to be able to construct this logical to physical mapping itself. It is clear, on the other hand, that the kernel must possess this ability, since otherwise the Start-I/O call would never operate properly.

The device scheduler must then somehow be given this physical address information. This extra information could be put into the locked-box itself. However, any physical address information placed in the locked-box for that purpose must be in addition to the logical address information previously contained in the locked-box, since the logical information is required by the Start-I/O call. Alternatively, this physical address information could be communicated to the scheduler by some other means.

If the needed information resides in the locked-box along with the rest of the original request, then the

to release the relocation registers which it has pointing at the lock-box area, then that is its prerogative -- it has just thrown away information which might have been useful. The scheduler cannot cause the kernel to start an incorrect shared device request merely by giving away its own access to the locked-box segment and reattaching something else in its place. The kernel's locked-box area, however, probably needs to be in "nice" 32 word blocks so that locked-box areas can be split up among various schedulers if one wishes to block the covert communications channel which might arise from allowing one scheduler to read another scheduler's locked boxes. [33] This effectively says the locked box segment is not a "real" segment in much the same way that the process communications buffer is not a real segment -- it is not the proper size for a real segment -- otherwise lock-box segments would rapidly eat up all available memory since they must remain locked into memory for the same reasons

---

[33]. It might be possible for the scheduler to communicate extremely limited bits of information to processes which use the device it schedules by the scheduling of the request -- which can be sensed by the process since it is notified when the request is both initiated and completed. The following might be possible: user X has written a disk scheduler with the property that it will start a request from process X only when device scheduler Y has a request for segment Z from process W. In this manner process X may be able to sense when W is using Z.

scheduler needs some means of accessing that portion of the locked box, so the question arises of where the locked-box should be stored. The most convenient place from the scheduler's viewpoint is somewhere within its virtual address space. However, as previously noted, it must reside in a portion which cannot be modified by the scheduler. [31] This would seem to be a rather strange segment unless it is itself part of the process communications buffer, since "normal" segments can be freely attached and detached, and it seems extremely unreasonable to make special cases of device schedulers "coded" into the kernel. [32] Since the kernel always "looks" at locked-boxes through its own space rather than the process space, the kernel will always read exactly what it placed in the locked-box. Thus if the scheduler decides

---

[31]. Effectively, some segment of the scheduler must be attached with read-only access in such a manner that it may never be attached with read/write access.

[32]. Special cases are "coded" into the kernel already in some sense, but not in the same manner. Certain process already possess special privileges. These privileges, however, are strictly a function of policy, as represented in the policy protection data, and can be changed fairly easily, although such changes may drastically reconfigure the manner in which the system operates. The sort of checks which might be necessary here would be on the order of: is process X a device scheduler? If so, then 1) automatically attach supervisor relocation register 6 to lock-box area F(X) with read-only access, and 2) never allow X to attach that area with read/write access. Actually 2) is not required as long as the kernel never allows any process read/write access to lock-box area F(X).

the kernel must be able to access them at all times. [34]

All of this mess can be avoided, however, if one does not put the physical address information the scheduler needs in the locked-box itself. If this information is compact enough (and one suspects it is or can be made so), then it can probably be squeezed into a normal send-message type message slot and handled similarly. If the requesting process possesses the necessary physical information [35] then it could send the message to the scheduler directly. If the process lies about the information, then the worst that could happen is that the device is scheduled incorrectly, since the kernel will do the proper thing regardless. The result of the scheduler being lied to about the physical address information by the user is precisely the same as if the kernel had misinformed it.

In fact, one can remove the burden of telling the scheduler about the locked box from the kernel entirely if

---

[34]. This would not have to be the case if the Request-I/O call fails if the lock-box segment is not core-resident and the Start-Indirect-I/O call fails likewise.

[35]. It may not, for the same reasons that the scheduler doesn't. However, the process might be expected to know much more about the segment, since the segment, after all, presumably belongs to it.

one lets the requesting process do this be means of a send-message to the scheduler, provided that one identifies where the request will be placed in such a manner that the requesting process can communicate that information to the scheduler in its message (so that the scheduler knows what to say to the kernel in the Start-Indirect-I/O call). If the process lies about the locked-box, the results are no more disastrous than if the scheduler picked a random locked-box and tried to do a Start-Indirect-I/O call on it, regardless of whether it had any reason to believe that locked box contained one of its requests.

The following conclusion can be made from the preceeding observations: where locked boxes are kept is irrelevant as long as 1) they can never be written by any process [36] , 2) a convenient way of naming locked-boxes is devised, and 3) the physical address information is sent to the scheduler along with the locked box name. Sending of the physical address information and the name of the locked box can be done either implicitly by the kernel in the Request-I/O call, or, explicitly via a Send-Message call by the requesting process. The kernel, however, probably possesses more accurate physical address

---

[36]. Not even the original requestor can be allowed to modify the locked-box, since the kernel determines the process identification of the original requestor from the contents of the box itself if that information is not implicit in the box's location.

information than the process and could relay this information to the requesting process as part of the return information in the request-I/O call. [37]

Although it slightly complicates the kernel, it is probably safer and more convenient to have the kernel notify the scheduler of the request and physical device information concerning the request automatically. This has the additional nice property that the code can be efficiently shared in the Swap-In and Swap-Out calls as well, effectively causing Swap-In and Swap-Out to look like a CPU scheduler Request-I/O call with a kernel constructed request.

---

[37]. This is another possible communications path, however, unless one checks to make sure that the requesting process possesses the required access to the segment at the time of the Request-I/O call, as well as at Start-I/O time. Otherwise Request-I/O calls could be used to sense the location of arbitrary segments. There are a number of arguments against requiring the process -- or even allowing the process-- to get at physical addresses of logical segments which are concerned with covert communication channels.

5.6.2 Start-Indirect-I/O Design

The Start-Indirect-I/O kernel call is the final step in the sequence by which processes which do no possess full access to a given shared device cause I/O to that device to be initiated in their behalf. The normal sequencing of operations here is a Request-I/O by process A to device B, eventually followed by a Start-Indirect-I/O call for process A executed by the process which owns device B.

Execution of a Request-I/O kernel call causes the request to be placed in a locked-box where it cannot be modified by any process. Then the owner of the device for which the request is made is informed that it has a pending request. Eventually the owner of the device decides to start the requested I/O and issues a Start-Indirect-I/O kernel call referencing the locked-box containing the request. The kernel first checks that the locked-box is a valid one and that the given device is not busy and is attached to the Start-Indirect-I/O requestor, then treats the request as if it had been a normal Start-I/O requested issued by the original requestor.

The Start-Indirect-I/O kernel call itself posed relatively few interesting design questions. Most of the interesting points concerning the initiation of indirect I/O operations occur with regard to the Request-I/O call

and have been discussed in the previous section.

Two points are of interest, however. The first concerns the type of errors which are (or could be) reflected back to the process attempting the Start-Indirect-I/O kernel call. [38] The requesting process can be held entirely responsible for two categories of errors: 1) Start-Indirect-I/O calls which are attempted while the given device is still busy, and 2) Start-Indirect-I/O calls which reference an invalid locked box. These two classes of errors can be totally attributed to the malfunctioning of the requesting process and hence are reflected back to that process.

There is a separate class of errors, however, for which the Start-Indirect-I/O requestor can not be held responsible -- namely those errors which are discovered when the actual Start-I/O call is performed. Errors falling within this third classification result not from errors in the Start-Indirect-I/O call, but rather from previously undetected errors in the request specified in

---

[38]. This process is usually the device scheduler for the shared device, but not necessarily: any process can "attempt" any kernel call; whether that attempt will succeed is an entirely different matter.

Request-I/O call. [39] The desire to reflect this last class of errors back to their original source (the original requestor) was the motivation behind placing kernel call return information in certain cases in the process queue rather than in fixed locations in the communications buffer. [40] This information cannot be trusted to be sent back to the requestor by the device scheduler via a Send-Message call since the device scheduler would no longer be a communications sink. [41]

The second point of interest concerns the grain of control which the initiator of the Start-Indirect-I/O call is to have over the operation of the device it is

---

[39]. These errors are undetected only because the kernel does not examine the contents of the request in any detail when the Request-I/O call is performed. No validation of the request is made in the Request-I/O call. Security checks made during the Request-I/O call cannot be assumed valid at the time the Start-Indirect-I/O call is performed. Hence the security checks must be repeated at actual Start-I/O time and so any validation of the request made during the Request-I/O call is redundant.

[40]. This phenomenon is discussed in more detail in section 3.6.

[41]. See [POPEK74A] concerning communication sources and sinks.

scheduling. The question to be answered here was whether it was necessary (or even desirable) to allow the Start-Indirect-I/O requestor to specify the same type of control options which would have been available had the request been a "normal" Start-I/O rather than an indirect one. More specifically, should the requesting process be allowed to specify whether it wishes to receive an interrupt when the I/O completes? The decision was made that a device completion interrupt would always be sent to the device scheduler for the shared device upon device completion. [42] Whether the original requestor (the process which did the Request-I/O call) also receives an interrupt upon the completion of the requested I/O depends on whether an interrupt was requested in the original request. [43]

---

[42]. This makes matters slightly simpler in the kernel, but not significantly.

[43]. It is doubtful that much use can be made of shared I/O without requesting an interrupt, however. A process which does not have the device attached cannot do status calls to poll for device completion. Thus if an interrupt is not requested in the original Request-I/O request, the process has no real way of knowing when the I/O has completed.

6. Conclusions

The overall design of the UCLA Virtual Machine System is now completed. A significant portion of the kernel has been flowcharted, coded, keypunched and submitted to heuristic debugging. The major pieces of the Virtual Machine Monitor have been written and tested. Several virtual machines, including the DEC DOS single user operating system, have been successfully run under the system, as well as several sets of "virtual" CPU diagnostics. The verification of the kernel code is underway.

In general the original design goals have been reasonably met: 1) already the system has proven useful enough to gather some interesting facts concerning "typical" operating system performance patterns; 2) although it is already fairly obvious that additional hardware assistance for the virtual machine monitor will be necessary before system performance is high enough to service a sizable user community, it seems likely that system performance will be more than tolerable with such modifications; [44] and 3) the system has been reasonably inexpensive, having gone from original design to near complete implementation in a relatively short period of time. It is not yet certain whether the primary goal of a verified system will be met, since the verification effort

is still underway. Yet it does seem quite likely that the verification task will soon reach a successful conclusion.

Two significant portions of the system, the initiator and the updater, still remain to be designed in detail. The kernel's underlying policy protection data structure (the structure interrogated by EVAL) still remains to be designed in detail. The mechanism by which the updater controls changes in this protection data and the necessary kernel support for such operations structure still remain to be designed. Several "convenience" changes in the way that capability and page faults are handled are currently under consideration and require detailed examination as to their feasibility and possible impacts on the system as a whole. [45]

Only time and grueling measurements will tell whether the initial system goals have been truly met. The success

---

[44]. This is not a system design fault, however, but instead is a direct result of the difficulties involved in virtualizing the 11/45 hardware.

[45]. These possible changes involve making capability faults totally transparent to the faulting process. The fault would instead be reflected to a special process (the CPU scheduler, perhaps) to be dealt with, and the faulting process would be blocked until the proper capability information is brought into memory.

or failure of the verification attempt will be the only true test of the system design. If it is successful, then much will have been learned, and if it is not, then much still will have been learned. Either way, the research will have been worth the effort put into it.

BIBLIOGRAPHY

1. [DEC73A] Digital Equipment Corporation. PDP 11 Peripherals Handbook, Digital Equipment Corporation, Maynard, Massachusetts, 1973.

2. [DEC73B] Digital Equipment Corporation. PDP 11/45 Processor Handbook, Digital Equipment Corporation, Maynard, Massachusetts, 1973.

3. [DEC74] Digital Equipment Corporation. Option Description: Virtual Machine Extensions to the PDP-11/45, Document No. CSS-MO-F-90-4, Digital Equipment Corporation Computer Special Systems, Maynard, Massachusetts, 1974.

4. [GOLDBERG74] Goldberg, Robert P. "Survery of Virtual Machine Research," IEEE Computer, Vol. 7, No. 6, pp. 34-45, June 1974.

5. [IBM1] International Business Machines Corporation. The Considerations of Data Security in a Computer Environment, International Business Machines Corporation Data Processing Division, White Plains, New York.

6. [PARNAS72B] Parnas, D. L. "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM, Vol. 15, No. 12, pp. 1053-1058, December 1972.

7. [PARNAS72A] Parnas, D. L. "A Technique for Software Module Specification With Examples," Communications of the ACM, Vol. 15, No. 5, pp. 330-336, May 1972.

8. [POPEK74C] Popek, Gerald J. and Charles S. Kline. "The Design of a Verified Protection System," Proceedings of the International Workshop on Protection in Operating Systems, Rocquencourt, France, August 1974, pp. 183-196.

9. [POPEK74B] Popek, Gerald J. and Robert P. Goldberg. "Formal Requirements for Virtualizable Third

Generation Architectures," Communications of the ACM, Vol. 17, No. 7, pp. 412-421, July 1974.

10. [POPEK74A] Popek, Gerald J. and Charles S. Kline. "Verifiable Secure Operating System Software," AFIPS Conference Proceedings, Vol. 43, pp. 145-151, 1974.

11. [POPEK75] Popek, Gerald J. and Charles S. Kline. "A Verifiable Protection System," Proceedings of the International Conference on Software Reliability, pp. 294-304, April 1975.

12. [RAGLAN73] Ragland, L. C. A Verified Program Verifier, Ph.D. Thesis, University of Texas, Austin, 1973.

13. [WULF74] Wulf, W. et al. "HYDRA: The Kernel of a Multiprocessor Operating System," Communications of the ACM, Vol. 17, No. 6, pp. 337-345, June 1974.

Appendix A: Thoughts on the Trojan Horse Problem

When one normally encounters authentication problems one generally is led to address the issue of identifying the user to the system. There is, however, a converse problem -- that of identifying the system to the user. While a considerable amount of attention is typically given to the former problem, little concern is usually given to the latter, yet the disasters which can result from faulty identification of the system to the user are often as horrendous (and perhaps even worse) as those which result from the faulty identification of the user to the system. While user identification flaws are often dangerous to the system as a whole, "system identification" flaws can be even more shattering to the individual user.

A typical scenario which gives rise to an instance of what has come to be termed the "Trojan Horse" problem arises as follows: A user walks up to a supposedly unused terminal and pushes a button. The terminal responds, asking him to log in by entering his user name, account, and password. The user does so and "appears" to become properly logged into the system. In reality, however, the terminal was not "unused" and the "system" to which he had identified himself was not the system at all, but rather a process masquerading as the system. The unwary user has now given that process his user name, account, and password

and may never suspect that his access rights have been compromised in any way.

What can be done to prevent the Trojan Horse problem? Two "simple" solutions come to mind: 1) The system can reserve certain characters or sequences of characters for itself so that no Trojan Horse process can output an imitation system herald; or, 2) the system can reserve certain characters or sequences of characters such that their occurrence on an input channel forces control to be returned to the system. Can either of these alternatives be reasonably implemented under the kernel?

Upon examining the first alternative, one finds that it is exceedingly easy, yet also exceedingly inefficient to have the kernel monitor all characters being output to user terminals for purposes of maintaining the integrity of a "system identification" character. There is no real efficiency problem if one restricts the class of terminals used by the system to to LA30-like devices (terminals which output in single character mode, rather than outputting characters stream or line-at-a-time). The kernel is already providing output character buffering there, for simplicity as well as efficiency. The problem becomes much more difficult to handle, however, if DH11-like devices (devices which do direct memory accessing on output) are to be utilized in any reasonably efficient fashion.

DH11-like devices require either that the kernel buffer all the characters to be output in the kernel's own data space, check them each for validity, and then perform the multi-character output on the characters from the kernel's space, or that the kernel validate all the characters while still resident in the process space, then point the device at them after first insuring that the given process will not be run again until after the I/O has completed. Otherwise the process could change the characters between the time they were checked and the time they were output-- clearly a security flaw if the inability of random processes to output the "system identification sequence" is a vital security guarantee of the system.

Secondly and more important from a security standpoint, the first method is not fool-proof. Although one may dedicate the system identification character for the exclusive use of the system and system processes, it is exceedingly difficult to prevent malicious processes from subverting a valid use of the identification character for its own devious use by clever terminal manipulations. If one dedicates a sequence of characters to the system (carriage return, followed by line feed, followed by @, for example), it is most difficult to prevent processes from outputting a character sequence which "looks" much like the identification string -- carriage return, line feed, null,

@, for instance, or line feed, carriage return, thirty-six nulls, backspace, @ -- sequences which are quite distinct from the system identification sequence as far as the operating system is concerned, but which are often indistinguishable to the user sitting at the terminal.

Single system identification characters (as opposed to character sequences) are slightly less prone to problems of the sort mentioned above but still fall prey to them in circumstances which are often dependent upon physical characteristics of the terminal in question: a single character identification scheme, for instance, is more likely to be subverted on a CRT-type and "intelligent" terminals where one typically has a large degree of flexibility in invisibly repositioning the cursor than on hard-copy devices. Subversion of both single and multi-character identification schemes in this manner rest, of course, on one's ability to find instances of the identification sequence legally output to the user terminal, something which is considerably much harder to accomplish in situation where a single character is reserved for all time as the system identification character. [1]

Much more useful than a "system identification " character provided by the first alternative, might be a system escape character offered by the second alternative.

Such a system escape character might be a single special character which would force the user terminal to be attached to a specific part of the system (the initiator, most likely, especially since it will be verified). This action is a likely candidate for kernel handling, and can be implemented with reasonable efficiency in the kernel as long as the kernel continues to buffer console input.

One would probably not find much sophistication in such an escape mechanism implemented in the kernel. One might expect the escape character to be a reserved character. When typed, it would always force the terminal to become attached to the initiator -- no backspacing or rubouts if typing it was accidental. [2]

---

[1]. There are considerable problems, however, in even choosing such a reserved character. It must be a visible character and one would wish it to be uniform for all terminals. These two requirements drastically limit the range of possibilities to the 'normal' upper-case only alpha-numeric character set plus a small number of extra symbols (essentially to a "typewriter" compatible subset of the available characters). This, in turn, limits the character echoing capabilities available to normal processes and one is likely to find oneself in the unfortunate situation of being forced to echo some other character sequence than the one typed by the user, at least in the instance where the reserved system character was input from the terminal. Such may well be the cost of security in this instance unless one is willing to go one step further down the road and eliminate the reserved system character from being input by the user in the first place, forcing the use of an escape sequence (which can be echoed exactly as input) which is interpreted by the software as an occurrence of the reserved symbol.

[2]. Such sophistication could be implemented by the initiator once it has received control, however.

Although the current implementation of the kernel supports neither of these alternatives, it appears that it would be relatively easy (and also relatively efficient) for the second alternative to be handled by the kernel.

Appendix B: Scenarios

Appendix B.1: A User Scenario

A typical user session on our system is intended to occur as follows: [1]

User X walks up to terminal Y and types the initiator escape character (control E, for example). The initiator responds with "UCLA VM System. Enter user name and password, please." The user types in his name, X, which the initiator echoes on the terminal, followed by a space and then enters his password, which is not echoed by the initiator, followed by a carriage return.

The initiator checks the combination of name and password against its list of legal name-password pairs, finds a match and responds by typing out the date and time, followed by "Enter a command or type HELP for help."

User X is already familiar with system commands and types "CREATE VM <cr>", to which the initiator responds with "ENTER NAME OF VM TO BE CREATED". X responds with "DOS <cr>". [2] The initiator then prompts with "DO YOU WISH TO SPECIFY SYSTEM CONFIGURATION?" X responds with "NO

---

[1]. The reader must bear in mind, however, that this merely represents a hypothetical sample session, since the entire system has not yet been completely implemented.

<cr>" and the initiator creates a DOS virtual machine with a default configuration and attaches console Y to the newly created VM.

The DOS herald appears on the terminal and user X logs into the DOS and does whatever it was he wanted to do. Some time later he has finished his DOS session and now desires to use the network rather than DOS. He communicates with his Virtual Machine Monitor, informing it that he wishes to be reattached to the initiator by typing %ATTACH <escape> (TO) INITIATOR <cr> (assuming the VMM escape character to be '%').

The VMM reattaches the terminal to the initiator, which in turn indicates its presence by prompting with "ENTER A COMMAND OR TYPE HELP FOR HELP". This time X types "ATTACH VM <cr>". The initiator responds with "DO YOU WISH TO DESTROY YOUR PREVIOUS VM<, (ES) OR N(O) ?" User X types "YES <cr>", the initiator destroys user X's previous VM (in this case, a DOS), and then prompts the user with "ENTER NAME OF EXISTING VM TO BE ATTACHED ". X types "ANTS <cr>" [3] and the initiator responds with "Attaching Teletype to ANTS".

---

[2]. DOS is a DEC single-user operating system.

The ANTS login prompt spews out onto the terminal and X converses with ANTS for a while, then decides to logout of the system entirely. He asks the VMM to reattach him to the initiator and then types "QUIT <cr>" at the initiator. The initiator types out the time and date, "X HAS LOGGED OFF", and is ready to respond to the initiator escape character once more.

The terminal is now free to be used by another user.

---

[3]. ANTS is an acronym for Arpa Network Terminal System, a system developed at the University of Illinois for providing user access to the ARPA network.

Appendix B.2: A Shared I/O Scenario

The following represents a typical scenario of I/O to a shared device (non-dedicated device) by a user process.

User X has logged onto the system in a manner similar to that previously explained and has been given a DOS as his virtual machine. He is interested in doing some editing, so he has started the DOS editor and has entered several hundred lines of text into the editor's in-core editing buffer. Now X decides he has no more text to input, so he tells the editor to write the contents of the buffer out to a file on disk.

DOS, being a virtual machine, thinks it has some number of disks at its disposal and attempts to allocate the new file somewhere on one of its disks. In order to do this, perhaps, it first reads some directory information from its (virtual) disk. So DOS loads some interesting numbers into its disk registers and attempts to read from the disk. This is not what happens in reality, however. When DOS attempts to load its disk control registers, a real hardware memory management fault occurs, causing the processor to enter the kernel at an entry point for user mode memory management traps. This memory management trap has occurred because the virtual machine monitor has insured that no real segment will ever be attached to the

user mode relocation registers which reference the upper 4K of the user's memory space (the region where the device control registers reside).

Upon being entered at the user memory management trap entry point, the kernel saves the contents of the memory management status registers in the shared kernel/process communications segment so that the virtual machine monitor can reconstruct what the virtual machine was attempting to do. After some amount of difficulty, the VMM realizes that the VM (DOS in this instance) really wanted to read from the disk, maps the memory addresses involved into segment name,offset and count, maps the disk address into disk segment name and offset and makes a Request-I/O kernel call.

The kernel constructs a locked-box containing the contents of the user request and sends it to the appropriate disk scheduler.

Some time later the disk scheduler decides to start the disk request and makes a Start-Indirect-I/O-Request kernel call to do so. At this point, the kernel starts the I/O request, destroys the locked-box, and informs both the VMM and the disk scheduler that the device has been started.

When the disk interrupt occurs, the kernel sends the disk status back to the VMM if it requested a pseudo-interrupt and notifies the disk scheduler that the disk operation has completed.

The VMM now reflects the completion of the disk I/O back to the VM via appropriate manipulation of the ready bit in the disk control register, possibly causing the VM to receive a virtual interrupt (if DOS was running with disk interrupts enabled).

Eventually DOS discovers that the disk is done, figures out where to allocate the file, and goes through the whole process again to write the file out on disk.

Appendix C: Input/Output Specification Terms

Functions

LEGAL-PROCESS-NAME(X) = TRUE

iff (there exists a process Y, such that) (X = Y AND

Y conforms to process naming conventions);

NO-IO-IN-PROGRESS(X) = TRUE

iff (there does not exist a device D, such that)

(DEVICE.USER(D) = X AND DEVICE.BUSY(D) = TRUE);

IN-CORE(X) = TRUE

iff (there exists a memory-frame M, such that)

(MEMORY-FRAME.NAME[M] = X);

RELOC-REG-CAN-BE-FREED(X) = TRUE

iff (X is not reserved for kernel/process communication);

ROOM-ON-DISK(X) = TRUE

iff (there exists a disk segment Y on disk X, such that)

(FREE(Y) = TRUE);

LEGAL-SEGMENT-NAME(X) = TRUE

iff (there does not exists a segment Y, such that) (Y = X)

AND X conforms to segment naming conventions;

```
DEVICE-INACTIVE(D) = TRUE
iff DEVICE.BUSY(D) = FALSE;

IN-SEGMENT-BOUNDS(X,Y) = TRUE
iff (X >= 0) AND (Y >= 0) AND (X + Y <=
 SEGMENT-BYTE-LENGTH);

LEGAL(D,A,C,O,S,R) = TRUE
iff (request R is a legal I/O request for device D);
FREE-LOCKED-BOX(R,P) = TRUE
iff (there does not exist a locked box index L and a device
 D, such that)
(LOCKED-BOX.REQUESTOR[L] = R ) AND (OWNER(D) = P)
AND (LOCKED-BOX.DEVICE[L] = D) AND (LOCKED-BOX.FREE[L] =
 FALSE);
```

Procedures

ALLOCATE-PROCESS(P):

FUNCTION:

Allocates a kernel process table entry for process P, initializing PC, PS, relocation registers, and giving supervisor portion proper access to its communications buffer.

```
BEGIN
   X <- NEW-KERNEL-PROCESS-TABLE-INDEX;
   PROCESS.NAME[X] <- P;
   PROCESS.PC[X] <- 0;
   PROCESS.PS[X] <- SUPV-PS;
   DO I = 0 TO 15;
      PROCESS.USER-PAR[X,I] <- 0;
      PROCESS.USERPDR[X,I] <- NO-ACCESS;
      PROCESS.SUPV-PAR[X,I] <- 0;
      PROCESS.SUPV-PDR[X,I] <- NO-ACCESS;
   END;
   PROCESS.SUPV-PAR[X,7] <-
      COMMUNICATIONS-SEGMENT.EXECUTABLE-PART[X];
   PROCESS.SUPV-PDR[X,7] <- READ-WRITE-ACCESS;
   PROCESS.SUPV-PAR[X,14] <-
      COMMUNICATIONS-SEGMENT.READ-ONLY-PART[X];
   PROCESS.SUPV-PDR[X,14] <- READ-ONLY-ACCESS;
   PROCESS.SUPV-PAR[X,15] <-
      COMMUNICATIONS-SEGMENT.READ-WRITE-PART[X];
```

```
    PROCESS.SUPV-PDR[X,15] <- READ-WRITE-ACCESS;
END; % OF ALLOCATE-PROCESS

DETACH-DEVICES(P);
FUNCTION:
Detaches any devices that might be attached to process P.
BEGIN
   DO I = 1 TO NUMBER-OF-DEVICES;
      IF DEVICE.OWNER[I] = P
         THEN
            DEVICE.OWNER[I] <- NULL;
            ATTACHED-DEVICES[P,I] <- FALSE;
      FI;
   END; % OF DO
END; % OF DETACH-DEVICES

DETACH-SEGMENTS(P);
FUNCTION:
Detaches all segments attached to process P.
BEGIN
   DO I = 0 TO 7;
      IF SUPV-RELOC-FREE(P,I) = FALSE
         ANDIF SUPV-RELOC-REG-CAN-BE-FREED(I)
         THEN
            SUPV-RELOC-FREE(P,I) <- TRUE;
            PROCESS.SUPV-PDR[P,I] <- NO-ACCESS;
```

```
            PROCESS.SUPV-PAR[P,I] <- 0;
            SEGMENT.REF-COUNT[ATTACHED-SUPV-SEGMENT[P,I]]
               <- * - 1;
            ATTACHED-SUPV-SEGMENT[P,I] <- NULL;
         ELSE
            PROCESS.SUPV-PDR[P,I] <- NO-ACCESS;
            PROCESS.SUPV-PAR[P,I] <- 0;
      FI; % TO SUPV RELOC CAN BE FREED TEST
   END; % OF SUPV DO
   DO I = 0 TO 7;
      IF USER-RELOC-FREE(P,I) = FALSE
         ANDIF USER-RELOC-REG-CAN-BE-FREED(I)
         THEN
            USER-RELOC-FREE(P,I) <- TRUE;
            PROCESS.USER-PDR[P,I] <- NO-ACCESS;
            PROCESS.USER-PAR[P,I] <- 0;
            SEGMENT.REF-COUNT[ATTACHED-USER-SEGMENT[P,I]]
               <- * - 1;
            ATTACHED-USER-SEGMENT[P,I] <- NULL;
         ELSE
            PROCESS.USER-PDR[P,I] <- NO-ACCESS;
            PROCESS.USER-PAR[P,I] <- 0;
      FI; % TO USER RELOC CAN BE FREED TEST
   END; % OF USER DO
END; % OF DETACH-SEGMENTS
SAVE-CONTEXT(P);
```

```
FUNCTION:
Saves the context of process P in the process table for the
process.
BEGIN
   DO I = 0 TO 7;
      PROCESS.USER-PAR[P,I] <- REAL-USER-PAR[I];
      PROCESS.USER-PDR[P,I] <- REAL-USER-PDR[I];
      PROCESS.SUPV-PAR[P,I] <- REAL-SUPV-PAR[I];
      PROCESS.SUPV-PDR[P,I] <- REAL-SUPV-PDR[I];
   END; % OF  DO
   DO I = 0 TO 5;
      PROCESS.REGISTER[P,I] <- REAL-REGISTER-SET-1[I];
   END; % OF REGISTER DO
   PROCESS.SUPV-SP[P] <- REAL-SUPV-SP;
   PROCESS.USER-SP[P] <- REAL-USER-SP;
END; % OF SAVE-CONTEXT

LOAD-CONTEXT(P):
FUNCTION:
Loads the context of process P from the process table  into
the hardware.
BEGIN
   DO I = 0 TO 7;
      REAL-USER-PDR[I] <- PROCESS.USER-PDR[P,I];
      REAL-USER-PAR[I] <- PROCESS.USER-PAR[P,I];
      REAL-SUPV-PDR[I] <- PROCESS.SUPV-PDR[P,I];
```

```
      REAL-SUPV-PAR[I] <- PROCESS.SUPV-PAR[P,I];
   END; % OF RELOCATION REGISTER DO
   DO I = 0 TO 5;
      REAL-REGISTER-SET1[I] <- PROCESS.REGISTER[P,I];
   END; % OF REGISTER DO
   REAL-SUPV-SP <- PROCESS.SUPV-SP[P];
   REAL-USER-SP <- PROCESS.USER-SP[P];
END; % OF LOAD-CONTEXT

SEND-SWAP-IN(REQUESTOR,SEGMENT,MEM-FRAME);
FUNCTION:
Constructs a swap-in request and transmits it to the
appropriate disk scheduler.
BEGIN
   DISK <- SEGMENT.SWAP-DEVICE[SEGMENT];
   MEMORY-SEGMENT <- MEMORY-FRAME.NAME[MEM-FRAME];
   REQUEST.SEGMENT <- SEGMENT.SWAP-SEGMENT[SEGMENT];
   REQUEST.OFFSET <- 0;
   SEGMENT.SWAP-IN-IN-PROGRESS[SEGMENT] <- TRUE;
   REQUEST-IO(REQUESTOR,DISK,MEMORY-SEGMENT,0,8K,
      SWAP-IN-ACCESS,INTERRUPT,REQUEST);
END; % OF SEND-SWAP-IN

RENOVE-ALL-BUT-SWAP-ACCESS(SEGMENT);
FUNCTION:
Updates kernel tables so that SEGMENT can be accessed only
```

```
in order to be swapped out.
BEGIN
   SEGMENT.CANNOTBE-USED[SEGMENT] <- TRUE;
   DO I = 1 TO NUMBER-OF-PROCESSES;
      DO J = 0 TO 15;
         IF PROCESS.ATTACHED-USER-SEGMENT[I,J] =
            SEGMENT
            THEN
               PROCESS.PREVIOUS-USER-ACCESS[I,J] <-
                  PROCESS.USER-PDR[I,J];
               PROCESS.USER-PDR[I,J] <- NO-ACCESS;
         FI;
         IF PROCESS.ATTACHED-SUPV-SEGMENT[I,J] = SEGMENT
            THEN
               PROCESS.PREVIOUS-SUPV-ACCESS[I,J] <-
                  PROCESS.SUPV-PDR[I,J];
               PROCESS.SUPV-PDR[I,J] <- NO-ACCESS;
         FI;
      END; % OF J DO
   END; % OF I DO
END; % OF REMCVE-ALL-BUT-SWAP-ACCESS

SEND-SWAP-OUT(REQUESTOR,SEGMENT,DISK);
FUNCTION:
Constructs a swap-out request for the segment and transmits
it to the disk scheduler for the appropriate device.
```

```
BEGIN
   REQUEST.SEGMENT <- SEGMENT.SWAP-SEGMENT[SEGMENT];
   REQUEST.OFFSET <- 0;
   SEGMENT.SWAP-OUT-IN-PROGRESS[SEGMENT] <- TRUE;
   REQUEST-IO(REQUESTOR,DISK,SEGMENT,0,8K,SWAP-OUT-ACCESS,
      INTERRUPT,REQUEST);
END; % OF END-SWAP-OUT

RELOC-SET(P,SEGMENT,R,ACCESS);
FUNCTION:
Sets the indicated relocation to point to the given segment
with the given access.
BEGIN
   IF R < 16
      THEN
         % USER RELOCATION REGISTER
         PROCESS.USER-PAR[P,R] <- SEGMENT.ADDRESS[SEGMENT];
         PROCESS.USER-PDR[P,R] <- ACCESS;
         ATTACHED-USER-SEGMENT[P,R] <- SEGMENT;
         USER-RELOC-FREE(P,R) <- FALSE;
      ELSE
         % SUPV RELOCATION REGISTER
         PROCESS.SUPV-PAR[P,R] <- SEGMENT.ADDRESS[SEGMENT];
         PROCESS.SUPV-PDR[P,R] <- ACCESS;
         ATTACHED-SUPV-SEGMENT[P,R] <- SEGMENT;
         SUPV-RELOC-FREE(P,R) <- FALSE;
```

```
   FI; % TO R TEST
END; % OF RELOC-SET

ZERO-DEVICE-STATUS(DEVICE);
FUNCTION:
Resets the device status for the device to a  well-defined,
initial state.
BEGIN
END; % OF ZERO-DEVICE-STATUS

ADD-ACCESS(P,A,O);
FUNCTION:
Updates the policy protection data so that P has  access  A
to object O.
BEGIN
END; % OF ADD-ACCESS

ALLOCATE-DISK-SEGMENT(SEGMENT,DISK);
FUNCTION:
Allocates segment SEGMENT on disk DISK.
BEGIN
END; % OF ALLOCATE-DISK-SEGMENT

ZERO-DISK-SEGMENT(SEGMENT,DISK);
FUNCTION:
Zeroes segment SEGMENT on disk DISK.
```

```
BEGIN
END; % OF ZERO-DISK-SEGMENT

FREE-DISK-SEGMENT(SEGMENT);
FUNCTION:
Deallocates disk segment SEGMENT.
BEGIN
END; % OF FREE-DISK-SEGMENT

WAKEUP(P);
FUNCTION:
Notifies CPU scheduler that process P might wish to run.
BEGIN
   COMMUNICATIONS.PROCESS-WANTS-TO-RUN[CPU-SCHEDULER,P] <-
      TRUE;
   COMMUNICATIONS.PROCESS-WANTS-TO-RUN[P,P] <- TRUE;
END; % OF WAKEUP

DEVICE-START(PROCESS,DEVICE,SEGMENT,OFFSET,COUNT,ACCESS,
   INTERRUPT-FLAG,REQUEST);
FUNCTION:
Initiates the requested I/O on the given device.
BEGIN
   DEVICE.REQUESTOR[DEVICE,ACCESS] <- PROCESS;
   DEVICE.MEM-SEGMENT[DEVICE,ACCESS] <- SEGMENT;
   SEGMENT.LOCK-COUNT[SEGMENT] <- * + 1;
```

```
   DEVICE.DEVICE-SEGMENT[DEVICE,ACCESS] <- REQUEST.SEGMENT;
   SEGMENT.LOCK-COUNT[REQUEST.SEGMENT] <- * + 1;
   IF INTERRUPT-FLAG = INTERRUPT
      THEN
         DEVICE.INTERRUPT-REQUESTED[DEVICE,ACCESS] <- TRUE;
      ELSE
         DEVICE.INTERRUPT-REQUESTED[DEVICE,ACCESS] <-
            FALSE;
   FI;
   REAL-DEVICE.MEMADDR[DEVICE,ACCESS] <-
      SEGMENT.ADDRESS[SEGMENT] + OFFSET;
   REAL-DEVICE.COUNT[DEVICE,ACCESS] <- COUNT;
   REAL-DEVICE.FUNCTION[DEVICE,ACCESS] <- ACCESS;
   REAL-DEVICE.INTERRUPT-ENABLE[DEVICE,ACCESS] <- TRUE;
   REAL-DEVICE.DEVICE-ADDR[DEVICE,ACCESS] <-
      SEGMENT.ADDRESS[REQUEST.SEGMENT] + REQUEST.OFFSET;
   REAL-DEVICE.GO[DEVICE,ACCESS] <- TRUE;
   DEVICE.BUSY[DEVICE,ACCESS] <- TRUE;
END; % OF DEVICE-START

SEND-DEVICE-STATUS(D,A,P);
FUNCTION:
Builds device status for the device and returns that
information to the process.
BEGIN
END; % OF SEND-DEVICE-START
```

```
LOCKED-BOX-INDEX(PROCESS1,PROCESS2);
FUNCTION:
Returns the index of the locked box allocated for shared
I/O requests from PROCESS1 to PROCESS2.
BEGIN
END;
```

STOPPED(P) = TRUE
iff (there exists an X, such that) (PROCESS.NAME[X] = P AND
PROCESS.STOPPED[X] = TRUE);

STOPPED(P) <- FALSE only if (there exists an X, such that)
(PROCESS.NAME[X] = P AND PROCESS.STOPPED[X] = FALSE);

FREE(P,R) = TRUE
iff (there exists an X, such that) (PROCESS.NAME[X] = P AND
PROCESS.RELOC-FREE[X,I] = TRUE);

Index