A SIMPLE AND FLEXIBLE SYSTEM INITIALIZATION MECHANISM

Massachusetts Institute of Technology
Laboratory for Computer Science (formerly Project MAC)
Cambridge, MA 02139

May 1977

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
HANSCOM AIR FORCE BASE, MA 01731

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
1400 WILSON BOULEVARD
ARLINGTON, VA 22209

LEGAL NOTICE

OTHER NOTICES

This technical report has been reviewed and is approved for publication.

William R. Price

WILLIAM R. PRICE, Captain, USAF
Techniques Engineering Division

Roger R. Schell

ROGER R. SCHELL, Lt Colonel, USAF
ADP System Security Program Manager

FOR THE COMMANDER

Frank J. Emma

FRANK J. EMMA, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Command & Management Systems

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-77-149	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A SIMPLE AND FLEXIBLE SYSTEM INITIALIZATION MECHANISM		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER MIT/LCS/TR-180
7. AUTHOR(s) Allen William Luniewski		8. CONTRACT OR GRANT NUMBER(s) F19628-74-C-0193 ARPA Order No. 2641
9. PERFORMING ORGANIZATION NAME AND ADDRESS Massachusetts Institute of Technology Laboratory for Computer Science (formerly Project MAC) Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS A023
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems Electronic Systems Division Hanscom AFB, MA 01731		12. REPORT DATE May 1977
		13. NUMBER OF PAGES 105
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

System Initialization
Layered System
Minimal Configuration
Core Image
Dynamic Reconfiguration

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents an approach to system initialization which is simple and easy to understand and, at the same time, is versatile in the face of configuration changes. This thesis considers initialization of a layered system and also considers the problems one might encounter in implementing the many dynamic reconfigurations required by this approach to system initialization.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

MIT/LCS/TR-180

A SIMPLE AND FLEXIBLE SYSTEM INITIALIZATION MECHANISM

Allen William Luniewski

May 1977

The research reported here was sponsored in part by Honeywell Information Systems Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641 which was monitored by ISTAO under contract No. F19628-74-C-0193.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

ACKNOWLEDGMENTS

A number of people and organizations have helped in the completion of this research and I would like to thank them now. I am sure to miss someone so I apologize in advance for these omissions.

First I would like to thank Dr. Clark, my thesis supervisor. His help has been very valuable in clarifying many of the ideas present in this thesis. I am especially grateful for the many hours he has devoted to reading earlier drafts of this document. Without his many helpful comments this thesis would never have been brought to completion.

Special thanks to Professor Liba Svobodova who took time out from a busy schedule to read an earlier draft of this thesis. Her comments were invaluable in clarifying the exposition of many of the ideas presented in this thesis.

I would also like to thank Professor Saltzer who suggested this line of research and whose continued help is greatly appreciated.

I would like to thank all of the members of CSR and especially Art Benjamin, Steve Kent and Jeff Goldberg. Their comments on various parts of this thesis have been helpful. Without the many interesting diversions they provided this research would have never been completed.

Lastly I would like to thank my family. My brother and sister, Tom and Cheryl, have been a comfort throughout the many years leading up to this research. To my parents without whose constant love and affection I would never have started or completed this research.

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

A SIMPLE AND FLEXIBLE SYSTEM INITIALIZATION MECHANISM *

by

Allen William Luniewski

ABSTRACT

This thesis presents an approach to system initialization which is simple and easy to understand and, at the same time, is versatile in the face of configuration changes. This thesis considers initialization of a layered system. The initialization mechanism is built upon three key concepts: existence of a minimal configuration, a core image of the system and dynamic reconfiguration. By assuming that the system will be running on the minimal configuration we generate a core image of the base layer of the system which, when loaded into core on any viable configuration, produces an operable base layer. As higher layers of the system are initialized dynamic reconfigurations of the lower layers are invoked to cause the system to run on the configuration actually present. The thesis also considers the problems one might encounter in implementing the many dynamic reconfigurations required by this approach to system initialization.

THESIS SUPERVISOR: David D. Clark
TITLE: Research Associate of Electrical Engineering and Computer Science

* This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on January 21, 1977 in partial fulfillment of the requirements for the degrees of Master of Science and Electrical Engineer.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter One

Introduction

Almost from the first appearance of the stored program digital computer there have been operating systems for these machines. The problem has always existed as to how to get an operating system, which has been designed for a class of machines, up and running upon a particular member of that class. This is a repetitive problem that occurs each and every time that it is desired to bring the system up after it has been down for a while. This is the problem of system initialization and is the subject of this thesis.

System initialization has been, for the most part, a neglected area of systems development. The techniques used by most current operating systems are either ad-hoc, difficult to understand and show correct or they lack versatility in the face of changes in the collection of hardware the system will be running upon. This thesis will attempt to develop a framework for system initialization that maintains this versatility but still is relatively easy to understand and show correct.

1.1 Initialization in General

To start we provide a general characterization of system initialization. To do so, we first make a few definitions.

We define the _hardware configuration_ to be the collection of hardware modules present in an installation as well as their interconnections (the system "wiring diagram"). For instance processors and memories are part of the hardware configuration.

The _software configuration_ consists of the values of various system parameters and the size of the system tables. For instance the maximum number of processes allowed on the system at one time is part of the software configuration.

We define the _configuration_ of a system to be the union of the hardware and software configurations.

With these definitions in mind we can make the following general observation about system initialization. Most operating systems are capable of running on a number of different configurations. The goal of initialization is to produce a version of the operating system tailored to a particular configuration and running upon that configuration. Most actions of initialization are present for exactly this reason. This view is supported by examination of many current operating systems including Honeywell's Multics, IBM's OS-360 and Control Data's SCOPE operating systems [CDC, Flores, HISIa].

The actual process of getting an operating system running on a collection of hardware has the following form. One (or potentially more) I/O device contains a storage medium, called the bootload medium, (1) that contains the programs and data necessary to bring up the operating system. In some system dependent way one or more processors begin running and use the bootload medium to get the operating system running on the particular configuration present.

We can identify three important times with system initialization. _System generation time_ is the time when the bootload medium is generated (created). This generally occurs during a previous period of the system's operation. _Initialization time_ is the period of time during which the operating system is being loaded onto the machine and initialized but before it is running normally. The time after the system is initialized, when it is running normally, is called _run time_.

This thesis will attempt to produce a simple and easy to understand overall system initialization mechanism. It is a fundamental premise of this thesis that an activity performed at system generation time or at run time is inherently simpler than the same action performed at

(1) For instance the I/O device might be a disk drive and the bootload medium a disk pack.

initialization time. We shall use this premise to produce our initialization mechanism.

1.2 The Need for Versatility

In choosing a way of achieving system initialization we want a method that is versatile; that is, it has the property that there is one version of the bootload medium that can be used on _any_ configuration to bring up the operating system. We will call an initialization mechanism that has this property configuration independent. For instance, if the system is initialized using magnetic tape as the bootload medium, we would like to be able to have one magnetic tape that can be used on any configuration to initialize the system. This versatility is very desirable, as the following example will show.

Consider a computer utility with a hardware configuration consisting of one processor and two boxes of memory and that we have a tape (1) specifically intended to bring up the operating system on this configuration. Now suppose that, as the system is running, one of the memories fails causing the system to shutdown (or, more likely, immediately "crash").

(1) For convenience we assume that the system is initialized using a magnetic tape. In principal any suitable type of I/O medium (such as disks) can be used as the bootload medium.

We will want to bring the system up as soon as possible so as to provide maximum service to the users of the computer utility. If it will take a long time, say days, to repair the memory, we are now faced with the problem of bringing the system up on this new, smaller, configuration of one processor and one memory. Our original tape cannot be directly used since the configuration has changed and is no longer the same as the one that the original tape was generated for. There seem to be four ways of getting our system up and running at this point.

First we might have previously generated a tape for this new configuration. If so we are in good shape and can just use that tape. This, however, is not in general likely since we experience combinatorial explosion in the number of tapes as the number of variables in the configuration increases. For instance, with two variables, each taking on two values, four different tapes are required to handle all possible configurations but five variables each taking on five values requires 3125 (=5*5*5*5*5) tapes. Thus, for all but the smallest systems, this technique will fail.

A second approach would be to go to the vendor of the system and ask him to generate, on his system, a tape for this new configuration. This is undesirable for two reasons. First it makes the availability of our computer utility dependent, in this case, on the availability of someone else's system (the vendor's). Second there might be a delay of hours before the vendor can supply the new tape. In either case we would experience a substantial delay in getting the system back up and running, violating a prime goal, availability, of a computer utility.

A third possibility is to use the original tape and then "patch" the system to reflect the new configuration. This is a poor way to proceed since the chance of an error while patching is very, very high. At best, this will result in a system that will not run at all; at worst the system will run but will operate incorrectly in an unnoticed way. Such undetected, incorrect operation is intolerable so we must also reject this approach.

The fourth possibility is a "starter" system. This is a separate operating system (possibly similar to the operating system that we wish to initialize) with the property that it can come up on _any_ viable (1) hardware configuration. This starter system is then used to generate a tape for this new configuration. This approach has two basic drawbacks. First the generation process may be a very long one. The resultant time delay may be intolerable. The second is that we may now have two operating systems to maintain, understand and show correct. We would like to avoid this added burden if at all possible.

---

(1) A viable configuration is a configuration on which the system can run. For instance a configuration consisting of no processors is _not_ viable.

We have seen four ways of getting our failed system up and running again. None of these schemes is completely satisfactory so we come to the conclusion that we must have an initialization scheme that can come up on any viable configuration if we are to achieve the goal of availability of our system. This thesis proposes an initialization mechanism that has this property of configuration independence.

The question naturally arises as to how present day operating systems address the issue of versatility in their initialization scheme. The answer, unfortunately, is that many do not. In the face of changes in the configuration many systems require a new bootload medium to be created. As this tends to be a long process this is undesirable for a computer utility. Some systems have other drawbacks beyond this. For instance IBM's OS-360 [Flores, IBMa] operating systems take a starter system approach where the starter system is just a version of OS made for a particular configuration. Unfortunately there exist configurations upon which one can run OS but which cannot run the starter system! On the other hand, Honeywell's Multics system has the property that one bootload medium can be used to initialize the system on any viable configuration (i.e. it meets our requirement of versatility). However, the method used, as we shall see in chapter two, is rather complicated and difficult to understand. In order to achieve this versatility a great deal of work is done at initialization time. This, however, is a time that, as we shall argue in the next chapter, is an undesirable one at which to perform complex operations. The goal of this thesis is to present a method for initialization which has the versatility of the Multics approach, but avoids its complexity.

1.3 Related Work

There is very little published material on system initialization. In [GM] a discussion of the initialization of the General Motors Timesharing System is presented. Initialization of IBM's OS-360 operating systems is discussed very briefly in [IBMa] and [Flores]. These provide a top level view of the goals and methods of achieving system initialization for these systems. The original design and motivation of Multics initialization is contained in [MSPM] in both a top level form and also in very great detail on a module by module basis. The original design is very close to the present implementation which is described in great detail in [HONa]. Unfortunately none of these documents and other documents this author has been able to find address system initialization in a somewhat higher, system independent, manner. Such a higher level view is one of the goals of this thesis.

This thesis builds upon the work done by Schell [Schell] in the area of dynamic reconfiguration. He discussed dynamic reconfiguration of processors and memory. In this thesis we will add to this work by including some aspects of the dynamic reconfiguration of I/O devices and various software reconfigurations.

The idea of layering of systems is an important one in this thesis. The concept of layering has appeared in numerous papers including [Dijkstra] and [SRI]. As we shall see in the next chapter this thesis only uses a very weak form of layering, which only requires that the bottom layer be always core resident; other forms of structuring a system, such as those in [SRI], [Reed] and [Huber], are equally amenable to the techniques presented in this thesis.

1.4 Thesis Outline: Preview of Approach

The ideas presented in this thesis have been inspired by the Multics time sharing system. As such they are directly applicable to that system. This does not mean, however, that the ideas cannot be applied elsewhere. In fact the method presented in this thesis should be applicable to any general purpose operating system that is based on a central processor - central memory hardware and that exhibits the minimal structure presented in chapter two. Its applicability to other architectures is, however, an open question.

In chapter two we present a model of a computer system. It is a top level view of the important aspects, from the point of view of this thesis, of the Multics system (hardware and software). We also look at the way in which Multics is initialized - an incremental mechanism. Using this knowledge we discuss the ways in which the scheme leads to difficulties in understanding Multics initialization.

Chapter three is a top level look at the initialization scheme proposed in this thesis. Initialization of a layered system is considered. We show that the hardest part of initializing a layered system is initializing the base layer. The proposed scheme to initialize a system attempts to take the extremely simple to understand core image approach to system initialization (in which an image of the system is just loaded into core to cause the system to run) and modify it so as to have a way of initializing the system that maintains the versatility which has been seen to be desirable.

The technique described achieves both simplicity and configuration independence by the combination of two concepts: a minimal configuration and dynamic reconfiguration. In reading chapters three through six the reader should keep in mind that the uniqueness of the approach presented in this thesis is in the combination of these two ideas to keep the simplicity of a core image approach and, at the same time, maintain configuration independence in our initialization scheme.

Chapter four describes the system generation procedure. It is here that the idea of a minimal configuration is explored in greater depth. By assuming the existence of a minimal configuration we see that many current initialization activities become actions performable at system generation time, with the result that we can create a core image of the base layer of the system.

Chapter five discusses the activities necessary to take the core image of the base layer of the system, load it into core and cause the base layer of the system to run. We also discuss the properties that the core image loader must have and problems associated with the size of the core image.

In chapter six we discuss how to initialize the second layer in the two layer system modeled in chapter two. It is here that the idea of dynamic reconfiguration is used extensively. Dynamic reconfigurations of the base layer are invoked as part of initializing the second layer to cause the base layer to be running using the full configuration actually present. We also see here that we only need one class of reconfigurations - additive. The subject of an initial paging area for the file system layer is discussed. The root of the hierarchial file system and storage system devices are discussed in detail.

The implementation of dynamic reconfigurations is discussed in chapter seven. Mention is made of the addition of processors and memory. The addition of I/O related hardware is discussed in detail. The dynamic changing of software parameters which control system operation is also touched upon. Lastly the problems associated with growing system tables, the major type of software reconfiguration, are discussed in detail. The subproblem of growing system segments is also discussed.

Finally chapter eight reviews the methods presented in this thesis. Some comments are made on the applicability of this method and possibilities for future research.

Chapter Two

A Model of a Computer System

In this chapter we will present an overview of the Multics operating system and some relevant aspects of the hardware it runs on. Our goal is to provide the reader with sufficient knowledge in these areas to enable him to appreciate the issues involved in system initialization on Multics. Using this knowledge, we then discuss how the hardware and software of Multics affects its initialization. The description of Multics serves as a general model of a two layer system and it is in the context of that model that the rest of the thesis will be presented.

2.1 Hardware Base

Although there are many aspects to the hardware that Multics runs on, for the purposes of this thesis we can abstract away from the actual hardware to a great extent. There are, in fact, only two aspects of interest: the system is centralized and the concept of a system wiring diagram is important.

This thesis only deals with centralized systems. These are systems consisting of one or more processors sharing memory and peripherals. Examples of such systems include Honeywell's Multics system, IBM's 360 and 370 systems, Control Data's 6600 and 7600 systems and DEC's PDP-10 systems.

The other important aspect is the concept of the system intermodule wiring diagram which reflects the physical interconnections between the various pieces of hardware, e.g. a processor or a memory module, that comprise the system. The system software needs to know this in order to direct commands from one module to another. For instance on Multics when a processor wants to initiate I/O it must know where, in the system wiring diagram, the I/O device in question is. Also, in the case of Multics, all intermodule communication is via system controllers, which also contain the memory, by sending messages along parts of the system wiring diagram.

2.2 Software Base

In this section we present a top level overview of the Multics supervisor, with the aim of presenting the structure of the system rather than implementation details. See [MAC73] for more details.

Multics is a general purpose timesharing system which implements a paged, segmented virtual memory, provides a hierarchial file system and provides for user controlled sharing of information. We will regard the Multics supervisor as a two layer system. (1) For the purposes of this thesis we shall regard each layer as being unlayered internally.

The top layer implements the file system. It is responsible for mapping user names of objects into segment identifiers. The rooted, hierarchial file system is implemented by this layer. This layer is also responsible for maintaining the attributes of segments such as the unique identifier, access control information and the creator.

The bottom layer, which we will call the base layer, provides the virtual machine that the file system layer runs on. It provides four basic functions. First, it includes the traffic control module which

---

(1) By layer we are referring to layering such as in Dijkstra's T.H.E. system [Dijkstra] or as in [SRI]. Layers 1 to i implement the virtual machine used by layer i+1.

implements processes, provides the interprocess communication mechanism and multiplexes physical processors among processes. Second, the paging mechanism and management of main memory are provided by this layer. Third, low-level input-output is the responsibility of this layer. It initiates all I/O and is responsible for determining the status of I/O operations. Fourth, this layer is responsible for fielding interrupts and faults (1) and directing them to their correct handlers. In this capacity it is also responsible for setting interrupt masks so as to prevent the occurrence of some, or all, interrupts.

This particular layering of the system has been chosen based upon three considerations. First the major criterion is to minimize the size of the bottom layer of the system. As we shall see in chapter four it is essential to make the bottom layer take up as little memory as possible. The second criterion is that the file system not be implemented in the base layer. The correct operation of the file system layer depends on the integrity of secondary storage. We do not wish the correct operation of the base layer to depend on this kind of external condition, as this would make it impossible to find a minimal configuration. For this reason we do not want the file system implementation in the base layer. The last criterion is simply one of convenience. The layering we have chosen models the Multics system very closely.

(1) A fault is a condition, such as overflow, that is generated internally by the processor receiving the fault. This is in contrast to an interrupt that is generated externally to the receiving processor.

2.3 Multics Initialization

In the previous sections we have provided a top level view of the hardware that Multics runs on as well as a simple view of the Multics supervisor. Using this knowledge, we will briefly touch on the issue: What makes current Multics initialization hard to understand? In answering this question we hope to provide further motivation for the remainder of this thesis.

In order to see what makes Multics initialization hard to understand, we must first get an idea of how it actually works. The following is a brief discussion; more detailed information is available in [HISIa].

Multics system initialization has been organized in a way so as to have one bootload tape that can be used on any configuration to bring up the system. Multics initialization has been organized so that almost all of the actions needed to produce a running system, as opposed to only the configuration dependent actions, take place at the time that it is desired to initialize the system. At the time that the bootload tape is generated all that is done is to take compiled programs and data and place them on the bootload tape.

The way in which the initialization of Multics occurs is best described by calling it an incremental mechanism. By this we mean that the total functionality provided by the supervisor and the environment (1) in which the supervisor runs are built up in an incremental manner. This means that while running in one environment, initialization makes another item of functionality work. It then proceeds to run in this new, augmented environment. In this way initialization builds its way from an initial, primitive, absolute addressing environment to the final environment consisting of a paged, segmented virtual memory with multiple processes.

Most of the initialization activities that Multics does are activities performed to produce a version of Multics adapted to a particular configuration. Unfortunately not all of them can be characterized in this way and we list some of them now for completeness. Some activities are the same for all initializations of the system no matter what the configuration is. In Multics the best example of this is an activity known as prelinking in which the external references of supervisor programs are statically resolved for the life of the system. It takes place at the same point of initialization, in exactly the same way, each and every time the system is initialized. Other such activities are the setting of system wide constants (such as page size

---

(1) We loosely define the environment of a module to be the collection of functions available to that module. At any given instant the environment of a module describes the total functionality currently available to that module.

and the size of various table entries). Other than these two items, all initialization operations can be viewed as activities geared to producing a version of the system adapted to a particular configuration.

This approach achieves its goal of one bootload medium for all configurations by delaying, as long as possible, configuration dependent decisions. All such decisions are made while the system is being initialized, when the full configuration is known. Initialization is taking the configuration information available to it at the time the system is being initialized and producing a version of the system adapted to this particular configuration and running on it. One can model what is happening by saying that the initialization algorithms and the bootload medium embody a model of what the system looks like on a general configuration, and the execution of initialization, on a particular configuration, uses this model to produce a version of Multics for the particular configuration present. However the method used to achieve this, the incremental mechanism, has problems as we will now see.

The incremental initialization mechanism serves to define a nested set of environments. It is important to note that the nested set of environments does <u>not</u> correspond to the layering of the system. Instead, at some point the current environment will correspond to that provided by a layer. The internal, amorphous environment of that layer will have been obtained by going through many nested environments. This nested set of environments tends to make initialization hard to understand in two ways.

First, it makes the understanding of the initialization routines themselves hard to understand. In order to understand whether or not an initialization program works correctly, it is necessary to know the environment that the program runs in. Thus to understand if an initialization program is correct one must first determine where in initialization it is called and the result (in terms of an environment) of all initialization programs that have run prior to it and then, finally, decide upon its correctness.

Second, normal supervisor routines are harder to understand. This is especially true for the base layer since, as we have noted, the base layer is essentially an unlayered collection of modules. As the base layer is being initialized, initialization uses features of this layer. This causes these supervisor routines to run in environments other than the <u>one</u> environment (the whole base layer environment) they normally run in. Thus to demonstrate the correctness of initialization one must show that these supervisor routines run correctly in not just one environment but in, potentially, many.

As an example of this last problem consider page control, the collection of modules which manage the multilevel memory system. When page control initiates a read of a page into core on behalf of some process, page control wants the current process to stop running and wait for the I/O to complete. In doing so, it abandons the processor to another process. However at the time page control begins running there are no processes because traffic control, the manager of processes, has not yet been initialized. The problem is to convince oneself that page control works in the absence of processes (or alternatively that traffic control does the right thing before it has been initialized). As it turns out, of course, it does work and it does so due to special casing inside of traffic control and the zeroing of core prior to the beginning of initialization.

2.4 Wrapup

We have seen a model of the Multics software as well as a model of its hardware base. The important hardware features are that it is a centralized, general purpose computer system and that knowledge of the system wiring diagram is necessary for the correct operation of the system. The Multics supervisor has been modelled as a two layer structure, each layer unstructured. The top layer implements the hierarchial file system while the bottom layer is responsible for I/O, interrupt handling, paging and the implementation of processes. The remainder of this thesis will use this model. Current Multics initialization has been seen to be an incremental mechanism and we have argued that it is this incremental character of initialization that makes it hard to understand. In the next chapter we propose an initialization scheme that is versatile, as is the Multics scheme, but which avoids the problems of the Multics incremental mechanism.

Chapter Three

Overview of the Initialization Scheme

In this chapter we will present an overview of our proposed initialization method. It works by taking the activities of the incremental initialization scheme presented in chapter two and ordering them so that they occur at very well defined times in well defined environments so as to avoid the discussed problems.

3.1 Initialization in a Layered System

The initialization of a layered system can be made simple by taking advantage of the layering present. Initialization will proceed upward in the system, initializing layer by layer, starting at the base layer, and continuing until the whole system is initialized. In this way the initialization task is broken into a number of disjoint parts.

We will discuss this initialization plan by considering the general case of a system consisting of many layers. We first initialize the base layer in whatever way seems appropriate and get it running. Then we initialize the second layer, while running on the virtual machine provided by the base layer, and get it running. Now, while running on the virtual machine provided by the second layer, we proceed to initialize the third layer. By proceeding in this way we can initialize the system layer by layer until the whole system has been initialized and is running.

After having initialized layers 1 to i, the system will be running on the virtual machine provided by layer i. We claim that this virtual machine provides sufficient functionality to initialize layer i+1. If this were not the case, the idea of walking up the layers, initializing as you go, would fail. This should not happen in a layered system where the virtual machine provided by layer i provides all of the functionality that layer i+1 needs to run. To see this, suppose that the virtual machine provided by layer i did not provide enough functionality to allow layer i+1 to be initialized. Layer i must then provide a "backdoor", for use only during initialization, which has the required extra functionality. Unfortunately there is no way for layer i to know for sure when initialization is over since such information would come from higher layers which are not trusted. Thus this backdoor is a defacto part of the virtual machine provided by layer i. For this reason the functionality provided by layer i to layer i+1 should be sufficient for the initialization of layer i+1. (1) We will assume that this is the case.

(1) An alternative would be to impose additional constraints on the system to the effect that only initialization programs may use, directly or indirectly, such backdoors. Another such constraint is, that when completed, the initialization program inform all layers that initialization is over, so that they may all shut the backdoors.

The writing of the initialization programs for layer i+1 is no harder than writing the programs that comprise layer i+1 since, in both cases, the programs will be running in the same environment - the virtual machine provided by layer i. Note how this favorably contrasts with many current initialization methods where the initialization programs run in a different environment than the regular system programs.

Thus, in a layered system, the hard part of initialization really comes down to the initialization of the base layer since it runs in the most primitive environment - that of the bare hardware. Higher layers run in progressively more sophisticated environments and thus are progressively easier to initialize. Even the second layer, in the system model presented in chapter two, sees a very sophisticated interface, one which includes processes and a paged virtual memory. The remainder of this chapter will primarily be devoted to outlining a scheme for the initialization of the base layer of a layered system.

3.2 Base Layer Initialization

We wish to produce a base layer initialization scheme that is simpler than the incremental mechanism presented in chapter two. The easiest way to simplify this, and any, mechanism is to make as much of it as possible go away. We shall take this approach.

In order to make as much possible of base layer initialization go away, we shall use a core image approach. A pure core image approach has the following form. At system generation time we create a copy of the base layer as it should appear in core when working. At the time the system is to be initialized, this copy (which we will call the base layer core image or core image for short) will be loaded into core. Since the core image represents a completely initialized base layer, the act of loading it into core and transferring control to it produces a running base layer. In the scheme presented below we will modify this so that only a <u>small</u> amount of initialization need occur after loading the core image in order to cause it to run.

We will take advantage of three other concepts: common activities, minimal configuration and dynamic reconfiguration. <u>Common activities</u> are actions that are the same for each and every initialization; i.e. they are configuration independent. An example might be the setting of a system wide constant such as page size. A _minimal configuration_ is a configuration, including both hardware and software aspects, which is guaranteed to be common to _all_ possible, viable configurations. One component of a minimal configuration would be the existence of, at the least, one central processor. _Dynamic reconfiguration_ is the changing of the configuration of the system, while it is running, in a way so as not to disrupt service to users. For instance in his thesis [Schell] Schell discussed the dynamic addition and deletion of processors and memories.

In later chapters we will discuss these three concepts more deeply but for now we will see how they, in combination with the core image concept, produce a useful, configuration independent system initialization scheme.

At system generation time we create a core image of the base layer by assuming that we will be running on the minimal configuration. Note how this contrasts with the starter system approach where a core image is generated for the configuration we would ultimately be running on. While creating the core image we perform all possible common activities. Note that we can only create the core image and find many common activities once we have assumed we will be running on a configuration. In our case we will have assumed the minimal configuration so that the initialization scheme is configuration independent. At system initialization time we take this core image and load it into core. At this point control is given to the base layer which must determine (or be told) the system wiring diagram corresponding to the minimal configuration since, as we will see in the next chapter, knowledge of the system wiring diagram is not assumed as part of the minimal configuration. The result is an operable base layer achieved in a very simple manner (a core image approach). Note that this core image must run since we have generated it assuming a configuration, the minimal configuration, _known_ to be a subset of the configuration actually present. The routines that initialize the next layer of the system, the file system layer, are now given control. The file system initializer, while initializing the file system layer, can now invoke any needed base layer dynamic reconfigurations to transform the configuration known to the base layer into the configuration actually present and desired. Realize that it is only the existence of these dynamic reconfigurations that allows us to maintain configuration independence in this initialization scheme. This is accomplished by the file system initializer invoking dynamic reconfigurations as needed.

This scheme is simpler than the incremental method since base layer initialization is reduced to, basically, a simple loading operation. Much of the hard work is embodied in core image generation which takes place in a "normal", well understood user environment at system generation time. The remainder of the work of initialization takes place in the form of dynamic reconfiguration. These reconfigurations are the same reconfigurations as used during normal system operation.

As such, since they are invoked in a normally running system, their use in getting the system running on the full configuration is, in some sense, not even part of initialization and, in any event, requires no additional effort to show correct once the system's regular operation is believed correct.

This scheme has not been used before for one very fundamental reason - the lack of a dynamic reconfiguration capability in most systems. Ultimately the success of this method and in particular the item which makes the assumption of a minimal configuration reasonable, relies upon the ability to perform many dynamic reconfigurations. Unfortunately most systems have little, if any, ability to perform dynamic reconfigurations. As a consequence this scheme could never even be considered.

In summary, our basic scheme is as follows. At system generation time we create a core image by assuming that we will be running on the minimal configuration and, at the same time, we perform all actions common to all initializations. When it is desired to initialize the system, the core image is loaded into memory to produce a running base layer. Dynamic reconfigurations can then be invoked to cause the base layer to be running on the configuration actually present. Initialization of higher layers can then occur.

3.3 Wrapup

In this chapter we have outlined how initialization in a layered system can proceed upward, layer by layer, through the structure hierarchy. In such a system the hard part of initialization is the initialization of the base layer. A core image approach to base layer initialization has been presented based upon the concepts of common actions, a minimal configuration and dynamic reconfiguration. The next three chapters will discuss each of the parts of initializing the system more deeply and explore the underlying concepts more closely.

Chapter Four

Core Image Generation

In the previous chapter we outlined our proposed initialization scheme. One of the cornerstones of this scheme is the ability to create, at system generation time, a core image of the base layer with the property that once loaded into core it is essentially functional. In this chapter we propose one way of generating this core image by assuming the existence of a minimal configuration.

4.1 The Process of System Generation

In order to generate the base layer core image, we will use techniques similar to those currently used by IBM's OS-360/370 and CDC's 7600-SCOPE operating systems. In these systems, a version of the operating system is produced that is tailored to the needs of a particular configuration. This is done by feeding the system generation procedures all the information that they need, such as how much memory and how much disk space will be around, the addresses of available main memory, device addresses, types and sizes of devices and the system wiring diagram. The system generation procedures then produce a version of the operating system made specially for the particular configuration described. It should be clear that an identical procedure can be used to produce a base layer core image for any system once we know the configuration. In our case we know that the configuration is the minimal configuration.

The output of the system generation process is the bootload medium which might be a disk pack or a magnetic tape. It consists of the generated core image, with possibly some information describing where it should be loaded at initialization time, the file system initialization routines as well as any data they need. In a more general case, at the time higher levels in the system are initialized it is necessary to have the routines and data that comprise them available to their initialization routines. They must, in general, be provided by the system generation process. For convenience we also place them on the bootload medium.

4.2 Where System Generation Occurs

The first question to answer is: Where do we generate the bootload medium and hence the base layer core image? System generation should take place in a standard user process, the same place that the system programmer does most of his work. Generating the bootload medium in a standard process has two very important advantages. The principle advantage is that the generation programs are written to run in the normal environment of a user process. This means that the developer of these programs is working in the environment where he does most of his work so that his task is eased by not needing to learn some new, and potentially unusual, environment for the system generation programs. As this environment tends to be well understood and well defined, the generation programs should be correspondingly easy to show correct. This choice, for instance, has been used in IBM's OS-360/370 and CDC's SCOPE [CDC] operating systems. The second advantage is a consequence of this one: most (if not all) of the system generation programs can be written in a high level language. The advantages of programming in a high level language are well known, so we will not repeat them here.

4.3 The Minimal Configuration

We say that configuration A is a subset of configuration B if the following are true:

i. The set of all hardware in A is a subset of the hardware in B.

ii. All of the system's hardware independent databases are smaller in A than in B.

Note that we do not include the system wiring diagram in our definition of subset. We also are assuming that a consequence of the first condition is that all of the system's hardware dependent databases (1) are smaller in A than in B. If we examine the set of all possible, viable configurations we assume that there will be one that is a subset of all of the others. We will call this configuration the minimal configuration. (2)

(1) A hardware dependent database is one that directly depends upon the hardware configuration.

(2) Current general purpose operating systems, built around central memory and central processors, seem to have this property. There may exist classes of architectures for which this is not the case. For these architectures we may not be able to define a unique minimal configuration. In this case our initialization scheme will not be directly applicable.

There are basically three aspects to the minimal configuration - processors, memories and the size of system tables. We will examine each of these in turn.

First we assume that the minimal configuration consists of one central processor. (1) One processor is needed or the system cannot run at all. Second and subsequent processors merely increase performance and reliability; they are not essential.

A processor without any primary memory is not very useful so our minimal configuration must contain some primary memory. We will make two assumptions about primary memory - its size and the existence of physical addresses. We will now elaborate on these two assumptions about primary memory.

4.3.1 Main Memory Size

We will assume main memory size based upon three considerations. First it is necessary to assume the existence of enough primary memory to contain the primary memory resident supervisor as it must, by definition, be in primary memory at all times. Second we must have some primary memory around to contain non-resident parts of the supervisor (i.e. parts of the layers above the base layer) and parts of user's

(1) This assumption is valid for traditional centralized architectures. For other architectures this may not be a valid assumption.

programs and data when they are needed, i.e. a paging pool for our virtual memory system. The size of this pool will be dictated by two considerations. The hardware will constrain a certain number of pages to be in core. For instance the pages containing the current instruction and the data it references may need to be in core. The system software may impose a lower bound on the size of the paging pool either through global constraints or per-process requirements. For instance, Multics requires that there always be ten free (unused but available for use) pages and, in addition, requires two pages to be in core, at a minimum, for a running process. (1) This, when coupled with the hardware constraints, will impose a lower bound of thirteen pages (2) on the size of the paging pool for Multics. Minimal performance considerations will cause the size resulting from these two considerations to be raised to the final minimal size of the paging pool. The third, and last, effect on memory size comes from the initialization process itself. If the core image is loaded into core by a software loader then there must be core for the loader. There must also be room for the code that ascertains the wiring diagram corresponding to the minimal configuration and for the file system initialization code. (3) These will all cause the minimal memory size

(1) The first page of the descriptor segment and the first page of the ring 0 stack.

(2) Ten free pages, two per-process pages and one for execution of the current instruction.

(3) Actually, as we will see in chapter six, only part of the file

to grow. Considering all three factors it is possible to determine the minimal size of main memory that is needed in order to bring up the system. Also knowing that memory comes in certain fixed sized chunks, we may be able to impose a still higher minimal size.

4.3.2 Main Memory Addresses

We will also assume the existence of the main memory addresses that the core image will occupy. This is necessary since there are many absolute addresses within the base layer, such as in segment descriptor words (SDW's) and page table words (PTW's), that must be filled in at system generation time in order to produce an operable base layer. If we do not assume these addresses it will be necessary for the core image loader to fill them in. This would entail the creation of a relocating core image loader. Such a loader is not, in general, a reasonable way to proceed. Let us see why.

system initializer will need to be loaded with the base layer core image so that its effect on minimal memory size will be small.

4.3.2.1 A Relocating Loader

A relocating loader would be responsible for taking the core image, in pieces perhaps, and loading it into available memory. As it does so, it must modify all of the physical addresses (as well as derived quantities) in the core image to reflect the actual physical addresses and not the ones assumed during system generation. On the surface this seems very reasonable since the construction of such a loader could follow the pattern of relocating loaders found on many present day operating systems for loading programs into a user's address space. The handling of explicit physical addresses, such as in SDW's and PTW's, is straightforward. However the handling of implicit addresses may be difficult. Consider a virtual memory system such as Multics. It is necessary for it to keep track of the status of each page of physical memory (usable/unusable, free/in-use ...). This is done by having an array, called the core map, describing the status of each page where the i'th entry describes the i'th physical page. To fill in the core map the relocating loader must be prepared to:

i. Fill in array entries describing the actual status of pages.

ii. Maintain linked lists of array entries.

iii. Grow the core map array to accommodate the pages actually used.

Item i. is self explanatory. For item ii., it may be necessary to maintain the entries describing free pages on a list. Similar lists might exist for used and unusable pages. Items i. and ii. are probably not that difficult to do. However allowing the loader to do them gives it a great deal of knowledge about the structure and contents of the core map; knowledge that we would like to keep only in the system so as to maintain system modularity. For item iii., suppose that at system generation time pages 0 to N were assumed to exist and corresponding array entries were allocated (i.e. the core image was generated to be loaded with no change into pages 0 to N). At the time the core image is loaded suppose that it is loaded into pages M to M+N (M>0). It is then necessary for the loader to allocate array entries to describe pages 0 to M-1. This will cause the array to grow. Although in principle possible to do, this may, as we shall see in chapter seven, be difficult to do. The loader must also take the entries in the core image core map and use that information to fill in the entries for pages M to M+N (where the system actually is) in the in-core core map. The net effect of all this is that it is possible to build a relocating core image loader but the loader would tend to get very complicated. Since it will run in a very simple and primitive environment, the bare hardware, such complexity is undesirable. Lastly, each of these tasks that a relocating loader would need to do, tend to give the loader a large amount of knowledge about the system and might have undesirable effects upon system modularity. For these reasons we reject the concept of a relocating loader and will, instead, just assume physical addresses.

4.3.2.2 Realization of Assumed Addresses

Having decided to assume physical addresses in main memory of the core image as part of the minimal configuration, we must now show that this assumption is not overly restrictive on possible configurations. By assuming the existence of physical addresses, an installation is required, at a minimum, to have those addresses realized in physical memory. To the extent that it is easy for an installation to assign these addresses to the available memory this is a reasonable approach. If, however, this assignment is difficult or impossible then the assumption of physical addresses will be overly restrictive on possible configurations.

Let us examine the hardware Multics runs on to see some of the problems that might arise. Each active module (processor or input-output multiplexor) in the configuration has an operator settable collection of switches that describe the base address of the memory attached to each port on the module. (1) The size of the memory on a given port is set by a plugboard in the port logic of the active module. The switches allow the operator to set the base address of a memory module as a multiple of its size. For instance suppose port 1 has a

(1) Each module has eight ports through which it communicates with memory modules.

256K memory attached to it, then the operator can set the base address of it to 0*256K, 1*256K, ..., 7*256K. No other base addresses are possible.

This technique has two drawbacks. First there is the chance for operator error. The operator must make sure that when he sets the switches on one active module, he also sets the switches on all other modules in the same way. (1) Failure to do so will either result in the bootload immediately failing or in the software detecting the error as the system comes up. The more serious problem concerns the inability to set the base addresses to the needed values without creating holes in the potential address space, potentially affecting future operation of the system. For instance suppose that we have two memories, one with 128K words and one with 1024K words. Also suppose that the 1024K memory is broken and that we have generated a core image that requires addresses 0 to 128K-1 to exist (i.e. the 128K memory as the low order memory will allow us to bring up the system). Clearly we can bring up the system by setting the base address of the 128K memory to 0*128K. However later, while the system is running, when the 1024K memory is fixed we will want to dynamically reconfigure it into the system. We will add it as having a base address of 1*1024K, the lowest available to it in the physical address space. The effect of this is that there is a hole in the physical address space - addresses 128K to 1024K-1 do not

(1) This is a requirement of the operating system and <u>not</u> the hardware.

exist. The only effect of this is in the core map array. Since it is an array we must allocate entries for the missing pages and mark them as unusable. Since this is a core resident table, these unused entries are wasting valuable memory resources. This loss of memory may be unacceptable.

If these drawbacks are felt to be severe, we propose a few hardware changes so that the assumption of physical addresses as part of the minimal configuration is reasonable. First the base address of the memory on a port can be made more flexible by allowing the base address to be set to be a multiple of some small number, say the smallest possible physical memory size. This will eliminate most chances of holes in the physical address space. Secondly the registers on active modules that reflect physical addresses should be software readable and settable. This includes not only the memory base addresses on ports but also a processor's interrupt and fault vector addresses as well as an IOM's mailbox address. (1) In this way the software can verify that all switches are set correctly and, if necessary, set them correctly and thus ensure correct operation even in the face of operator error. With these changes the assumptions about the existence and location of main memory are quite reasonable.

(1) An IOM is the programmable controller of I/O devices; i.e. it executes channel programs. Other names for this device might be I/O controller or channel. A mailbox is used as an incore communication area between processors and IOM's.

4.3.3 System Table Sizes

The last component of the the minimal configuration is the size of the various system tables. At system generation time it is necessary to allocate space for, and fill in, the system tables. Here we discuss the allocation issue, the filling issue will be discussed later. At system generation time we must decide the minimum size of the various system tables. We can do so based upon two considerations. First the actual system structure and design will force the tables to have a certain minimal size. For instance on Multics there will always be at least two processes around (1) which, as a consequence, requires that the active process table have at least two entries. Another example is the active segment table. (2) At least one entry is required for each always-active system segment. In addition, the implementation of the hierarchial file system requires the allocation of other entries. Minimal sizes can also be forced upon tables as a result of the assumed minimal hardware configuration. For instance if we have a processor table, we know that it must have one entry for the one, assumed processor. A better example is the core map; for each page of memory

(1) The idle process for one CPU and the initializer process.

(2) The active segment table contains the page tables for currently addressable segments.

that we have assumed exists, one entry needs to be allocated in the core map. For many tables the minimal size will turn out to be the empty, or null, table. An example of such a table would be the paging device map, which describes the status of pages on the paging device, since we have not assumed the existence of a paging device in the minimal configuration. It should thus be possible to decide, at system generation time, upon the minimal size of all system tables.

4.4 Common Actions

There is a collection of actions common to all initializations that can be performed at system generation time. These are either actions that are the same for every initialization regardless of the configuration or they are actions that can be performed at system generation time given that the system will be running on the assumed minimal configuration. We will call these actions common actions or configuration independent actions.

One such common action is the prelinking of the supervisor. The set of segments that comprises the supervisor does not change for the life of the system. Thus it is possible, at system generation time, to assign segment numbers to every supervisor segment. Having assigned segment numbers, it is then possible to resolve all external references within the supervisor. At the same time it is possible to create and initialize any data bases needed to enable base layer programs to actually make external references (they will have been resolved). (1) Then at the time the base layer core image is loaded it will immediately be possible for it to successfully make external references.

At system generation time we either know, statically, the size of each supervisor segment (for instance segments containing executable code are statically sized) or we can calculate it (for instance variable sized tables, making for variable sized segments, now have a known length due to assumptions made as part of the minimal configuration). The knowledge of segment sizes, when combined with our assumptions about main memory addresses, allows us to decide where in core each segment should be. We also can allocate, and fill in, an active segment table entry (ASTE) for every paged supervisor segment. For every supervisor segment we can then fill in its segment descriptor word (SDW), either pointing it to the segment itself or pointing it to the segment's page table (in its ASTE). Filling in the page table for paged segments and filling in the SDW for unpaged segments requires us to know the actual main memory address of these segments. Since we have assumed the existence of main memory addresses it is possible for us to assign these addresses at system generation time. As we assign these addresses we can also fill in the core map.

(1) In the case of Multics these are the combined linkage sections and the linkage offset table.

When the base layer begins running it will have a certain number of existing processes. The number and nature of these processes is known at system generation time. We can thus create and initialize, at system generation time, all of the per-process segments for these processes. Also we can allocate, and fill in, the active process table entry (APTE) for each process. This per-process initialization is done in a way that places each process in a known, desired state at the moment the system is running thus allowing the use of processes from the moment the system is loaded.

A large class of common actions come under the heading of table initialization. Since we wish to create a core image that can be loaded to produce a running base layer, it is necessary to create and initialize all of the base layer's databases. We have already described how we can fill in three major databases - the core map, active segment table and active process table. The remaining databases can be initialized in either a minimal or null state. For instance the paging device map can be initialized to show no paging device. The various databases that refer to peripherals (such as teletypes, disks and magnetic tapes) can be initialized to show that no peripherals exist. Note that these various table initializations are only possible once we have assumed the minimal configuration.

The last set of common actions consists of the setting of software parameters to some initial value. These parameters are used to control the actions of the software. For instance it may be possible to turn metering on or off by setting a software switch. It may be possible to turn system debugging actions on or off. Scheduler parameters must be set to some initial value. All such parameters should be set to some initial value so that the system may run correctly when loaded.

4.5 Wrapup

In this chapter we have described the system generation process. The system generation procedure runs in a standard user process and creates a core image of the base layer. This core image is generated by assuming a minimal configuration and performing all possible common actions. The actual generation process occurs in a way similar to that used by systems such as IBM's OS 360/370 and CDC's SCOPE. System generation then places this core image and the programs and data of higher layers on the bootload medium.

Chapter Five

Base Layer Loading and Initialization

In the last chapter we described the system generation procedure. As part of its output it produced a core image of the base layer on the bootload medium. In this chapter we describe how the core image is loaded into core and initialized to produce a running base layer. We see that the loading and initialization of the base layer are very simple operations.

5.1 The Core Image Loader

The core image loader is responsible for loading the base layer into core and giving it control. This loader is also responsible for validating that the core image has been loaded correctly.

The basic function of the loader is to take the core image from the bootload medium and load it into the place in core where the core image wants to be. The location where the core image must be loaded is, in general, variable (since it is a property of the core image) and should not be built into the loader. Once the core image is correctly loaded the loader must then give control to the core image.

The first question that arises is: How does the loader get the core image from the bootload medium into core? The loader fabricates, or has built into it, a series of commands for the I/O device containing the bootload medium to cause the core image to be transferred into core. This transfer represents the entire loading operation. Now we must see how the loader knows which I/O device contains the bootload medium. We can regard the loader as running on a configuration consisting of one processor, all of main memory and the bootload I/O device. (1) Thus it performs its I/O on the _only_ I/O device it has. The choice of I/O device is independent of the loader, he has no choice. Typically the choice will either be built into the hardware or will be settable by the operator via console switches. Note that this model fits very well with current hardware bootstrap loaders which have built into them a small program to read from an I/O device which is specified by operator settable switches.

Unfortunately it is not sufficient just to load the base layer core image into main memory and then let it run. At the very least it is necessary to check that the base layer has been loaded correctly. If for some reason the loading operation has not been done correctly the proper operation of the base layer is in doubt. In particular we would like to be sure that the data generated as the core image is the same

(1) The bootload I/O device is the I/O device containing the bootload medium.

data actually read into core. (1) This is a general problem in using mass storage media and its solution, in general, is beyond this thesis. To minimize the chance of such errors we propose to use two techniques. First, the bootload medium should be written in a standard data format thus giving give the core image loader the advantage of all the error detection (and possibly error correcting) machinery associated with standard formats. (2) This machinery will tend to prevent an erroneous core image from being loaded and not being detected. This technique, when coupled with standard hardware error detection on the I/O device, will reduce the number of undetected errors to a very small number. Second, to reduce this number even further, if that seems necessary, we can have the base layer perform checks upon itself once it gains control and before it passes control onto the initializer for layer 2. The simplest such check would be to compute a checksum on the whole core image. Other checks, such as data base consistency checks, could also be incorporated. Choosing which additional checks the base layer should make must be based upon the probability of an undetected error (without more checks) traded against the additional complexity these checks create before we can consider the base layer to be running. These two techniques, standard data storage formats and base layer self checking,

(1) Another form of data integrity is insuring that the bootload tape is only used by authorized personnel in authorized ways. This is a security issue that we do not address in this thesis.

(2) For instance, a Multics Standard Tape contains a checksum on each record to aid in detecting errors.

should reduce the number of undetected errors in loading the base layer to a small, and hopefully negligible, number.

The question naturally arises as to whether the loader should be in hardware or in software. For the purposes of this thesis it does not matter; it only matters that the result of the load operation is correct. From a practical point of view the necessity of handling variable requirements (load point and data formats) indicates that a software loader may be most appropriate. Using a software loader does, however, introduce the problem of how to load the software loader. This can be solved by either applying the results of this thesis recursively or by using a hardware loader. Note that ultimately a hardware loader will be used to get things going.

5.2 Core Image Size

In the last section we have implicitly made the assumption that the base layer core image (plus, potentially, the loader and the file system layer initializer) will fit into core. This section will address the possibility that the base layer core image (plus loader and file system initializer) is too large and does not fit into core.

The situation we are hypothesizing is one in which the configuration we are coming up on has sufficient memory for the proper operation of the system, however the core image (plus loader, etc.) does not fit into core. The effect is that the system could, in principle, run but cannot be initialized under our proposed scheme.

In this situation, there is really nothing we can do. The loader, when noticing the lack of memory, must simply stop initialization and report the problem to the operator - the installation has insufficient memory under our initialization scheme.

The question now is whether the situation of the base layer core image not fitting in core can reasonably occur. A system that had this property would be swapping parts of itself to and from secondary storage as required. Note that in such a system the swapping routines, at the least, must always remain in core. They cannot depend upon any swappable routines since at the point that the swappable routine was needed by the swapping routines it might be on secondary storage. This means that the swapping routines, and the routines they use, are self-sufficient and always in core. The self-sufficiency makes them a layer and, since they are always in core, we can regard them as a base layer that *must* fit in core. We can thus use the swapper as the base layer of the system.

We are now left with the possibility that the core image fits but the file system initializer plus, potentially, the loader do not. Both of these can be made very simple programs since their tasks are easy. As such they will take up little room and we will assume they will fit. From a practical point of view if the loader and file system initializer do not fit then the system is very much short of memory and will not be very useful. In any event with the price of memory dropping as it is this should not be a problem in the future.

5.3 Base Layer Initialization

Our technique of generating the core image creates the task of core image initialization which is not present in a pure core image approach. Two types of initialization must be performed. The configuration must be validated as being a superset of the minimal configuration and the system wiring diagram must be ascertained.

We have generated the core image based upon a number of assumptions about the configuration we will be running on. The correct operation of the base layer, which is of course our goal, depends upon our assumptions being correct. If we try to run the base layer upon a configuration that is not a superset of the the one it was generated for, we cannot, in general, guarantee correct operation. Thus core image initialization must verify that the particular configuration it is running on is a superset of the one it was generated for. For instance, depending on what was assumed in the minimal configuration, this might involve checking for the existence of disks, memory, front-end processors, central processors and other I/O devices. If we have assumed something about the system wiring diagram we must also verify it. A mistake here must almost surely be regarded as a fatal error causing initialization to fail. (1)

Note that when we discussed the minimal configuration we did not assume any knowledge of the hardware wiring diagram. This was quite deliberate since to assume this knowledge would be overly restrictive. If we were to assume any knowledge of the system wiring diagram we would, potentially, be restricting greatly the set of possible configurations that satisfy the assumptions making up the minimal configuration. (2) Since we have not assumed any knowledge of the system wiring diagram, and since this knowledge is essential for correct base layer operation, we must perform some actions once the core image is loaded and control given to it in order to provide this knowledge. We do not have to ascertain all such knowledge but, rather, we must only determine that part of the wiring diagram which corresponds to the minimal configuration. Having performed this initialization, we will have an operable base layer and control can be given to the file system layer initialization routines.

---

(1) Here we are allowing for an assumed configuration at system generation time other than the minimal configuration presented in the last chapter. For that configuration this phase disappears since the correct termination of the loader is sufficient to insure that the minimal configuration is present.

(2) If we were to try to assume knowledge of the system wiring diagram in the minimal configuration we would find that it would not be possible to find a minimal configuration.

Note that base layer initialization runs on the same virtual machine that the base layer runs on; quite probably this is the bare hardware. It is due to the primitive nature of this virtual machine that we have caused base layer initialization to be very small by taking the core image approach.

5.4 Wrapup

In this chapter we have seen how the core image is taken by the core image loader and loaded into core. We have seen what properties the core image loader must have. It has been argued that the base layer must always fit into core. Finally we have discussed the actions that must be taken to actually initialize the core image to produce an operable base layer.

Chapter Six

File System Initialization

In the last two chapters we have described how to produce an initialized, running base layer. This chapter will discuss the manner in which the next layer in the system, the file system layer, can be initialized. Particular attention is given to storage system devices and the root of the hierarchial file system.

6.1 Dynamic Reconfiguration

After the base layer is loaded and running normally it will think that it is running on the minimal configuration. Any hardware beyond that in the minimal configuration will not be in use. Various software tables may be smaller than that desired by the installation. The software parameters may not be set in the way the installation desires. It is thus necessary for the system to change itself to conform to the actual configuration. This process of changing from one configuration to another is known as _reconfiguration_; it is _dynamic_ _reconfiguration_ since it occurs while the system is running.

For our purposes, we need only look at "ADD" type reconfigurations; that is, we always add a piece of hardware (never delete) or grow a table (never shrink it). Our reconfigurations take this form due to our assumption of a minimal configuration. Since we start at the minimal configuration we can only need, by definition, to add hardware or grow tables in order to get to the actual configuration. This is fortunate since, as Schell points out in his thesis, this sort of "DELETE" type reconfiguration tends to be harder to implement then "ADD" type reconfigurations due to the necessity of breaking bindings in the "DELETE" case.

6.2 Reconfiguration

In this section we briefly list the kinds of reconfigurations that we will need. Reconfigurations fall into two basic categories - hardware and software. It should be noted that some hardware reconfigurations cause software reconfigurations as a side-effect (for instance, adding a memory may cause the core map to grow).

There are a relatively small number of needed hardware reconfigurations. We need the ability to add the second (and subsequent) processor in a configuration. We also need the ability to add additional boxes of memory if they are available. The ability to add individual pages of a memory will also be needed in order to use any pages that are part of a memory that the core image is loaded into but which are not actually part of the core image. Input-output multiplexors (IOM's) (1) will have to be added in order that the system can control I/O devices. It will be necessary to add I/O devices so that the system can communicate with the outside world.

A special case of an I/O device is a storage system device. These are storage media, such as disks, that contain the file system. The addition of such devices is really a two step process. First the I/O device itself must be added. This is a base layer reconfiguration. The effect of this is to open the communication path between the operating system and the I/O device. The file system must then verify that the contents of the device conform to file system standards. In particular, a previous crash may have left the volume in an inconsistent state, necessitating a salvage to get it in a consistent state, (for more information see, for instance, [Stern]).

Software reconfigurations fall into two basic categories. First it is necessary to be able to set the various software parameters to the values desired by the installation. In particular, debugging options must be chosen, metering turned on (or off) and tuning parameters set. The second category of software reconfigurations needed are table expansions. It is necessary to expand the initial, minimal tables

(1) An IOM is the programmable controller of I/O devices; i.e. it executes channel programs. Other names for this device might be I/O controller or channel.

created at system generation time to the size needed, or desired, by the installation. In particular it will be necessary to expand, at a minimum, the active segment table, active process table, core map, paging device map and physical volume table. (1)

The reconfigurations mentioned above, all of which are of the "ADD" type, are representative of the reconfigurations needed by this approach to system initialization. Here we have just listed these needed reconfigurations; in the next section we will see how the file system initializer uses them and in the next chapter we will consider some of the issues surrounding the actual implementation of them.

6.3 File System Initialization

File system initialization is basically a six step process. When complete the file system layer of the system will be running and initialization can proceed to initialize higher layers of the system. All of file system initialization runs on the virtual machine provided by the now running base layer of the system in order to avoid the problems we saw in chapter two.

(1) The physical volume table has an entry in it for each disk pack in the system.

At the time file system initialization gets control it sees a hardware configuration consisting of one processor and some memory. The configuration contains no I/O devices and, in particular, does not contain the I/O device that contains the bootload medium that, in turn, contains the remainder of file system initialization as well as the file system layer itself.

The first task of file system initialization is to gain access to the bootload medium. To do so a dynamic reconfiguration of the base layer is invoked to add the IOM that has the bootload I/O device on it. Next the base layer is invoked to add the bootload I/O device to the system. The effect of these is to open a physical communication path between the bootload medium and the operating system. These actions were necessary because knowledge of the bootload I/O device was not assumed as part of the minimal configuration. Also note that the core image loader <u>did</u> have this knowledge and, quite probably, passed this knowledge along to the base layer initializer. Up to now, however, this knowledge has neither been used or <u>needed</u>.

The next step is to acquire an initial secondary memory paging area. Up to this point page control (which provides the paged virtual memory), although operative, has not been very useful since it had no place to which to move pages when evicting them from core. In this state the system has been very memory constrained. By acquiring a paging area the system will no longer be memory constrained. The acquisition of a secondary memory paging area is necessary since the file system layer is large and will not, in general, be able to fit in core all at once.

At this point it is necessary to read the rest of the file system initialization routines as well as the file system layer routines into memory. (1) The necessity of getting the file system routines is clear; however, the need for additional file system initialization routines may not be obvious. Recall that the first part of the file system initializer must be loaded as part of the core image. Due to the desire to limit the size of the core image we are forced to keep the initial part of the file system initializer small. For this reason the initial part of the file system initializer only performs the above tasks. This second part of the file system initializer performs the remainder of file system initialization, summarized below.

The next step in file system initialization is to set up any databases needed by the file system. Since we are dealing with a rooted hierarchy it is necessary to find the root of the file system and verify its contents. Finding the root may involve invoking base layer dynamic reconfigurations to add an IOM or an I/O device. Finally it may be necessary to verify the contents of all, or part of, the rest of the hierarchy. At this point the file system layer is initialized.

(1) Recall that we placed the file system routines on the bootload medium for convenience. If the file system layer is elsewhere we look for it now and add that device.

It may also be desirable, but not essential, to perform some other reconfigurations during file system initialization. The operator's console may be added in order to allow communication with the operator; a paging device and more main memory might be added to improve performance; if multiple processes are present, the addition of more processors may improve performance. It is important to note that all of these reconfigurations being used are the same reconfigurations used while the whole system is up and running - they are not initialization time special cases as they were in the incremental initialization scheme.

In the next two sections we examine some of these aspects of file system initialization more closely.

6.4 The Initial Paging Area

As we saw in the last section it is necessary for the file system initialization routines to obtain a paging area. The normal paging area is on a secondary storage device which is part of the file system. However a system failure during a previous period of the system's operation may have left the contents of a storage device in an unreliable state. In particular we cannot be sure which records on the device are currently in use meaning that we cannot write to the device since valid data might be destroyed. Thus, the contents of the secondary storage device must be validated before being used as a paging area.

The validation of the contents of a storage device, however, is a file system layer function, normally performed while the file system layer is running. At file system initialization time we must perform this validation in order to get the file system running thus creating a problem. We cannot use the normal validation routines, since using them would cause them to run in an environment other than our normal one, and this, as we saw in chapter two, is not a good idea. Further, the validator tends to be a large, complicated program and will not, in general, fit into core all at once. Thus it must be paged in and out of core by the virtual memory mechanism. But we do not currently have an area to page to since that is what we are trying to get! This is a dilemma that can be solved in one of two ways.

Since we cannot use the normal validation routines, we must have an initialization time routine. We would like to keep this routine simple so as to keep file system initialization simple. We are going to argue that the initialization time validator need only validate that the storage system device is formatted properly and need not actually verify the contents.

If it is possible to page from the bootload medium we can just use it since we know its contents are correct. This might be possible, for instance, if the bootload medium is a disk pack.

Another, more general, solution is to reserve an area on secondary storage for the use of the file system initializer. This area will only contain information that is "recreated" each time the system is initialized (i.e. the file system layer and initializer) and not permanent data (i.e. user files). We can use this reserved paging area as the initial paging area each time the system is initialized since it will only contain the file system layer from the last initialization, which is no longer needed.

If we reserve a paging area on a storage system device for the initial paging area, the file system initialization verifier need only confirm that that particular device seems to be laid out properly before using it for the initial paging area. Checking the device format probably involves checking the volume label for proper contents and checking that the reserved area on the device does not overlap with the areas reserved for permanent files and that the records in each area actually exist. Basically these checks are to ensure that the reserved paging area is safe to use (i.e. permanent data will not be destroyed). These checks are very simple and doing them in a special initialization routine is no great hardship.

6.5 The Root

The file system layer is implementing a hierarchial, rooted file system. In such a file system there is one object, the root, that is the ancestor of all other objects. In order to operate correctly the file system layer must know where the root is and it must also believe its contents. Thus, as part of initializing the file system layer, we must also gain access to, and validate the contents of, the root.

This requirement creates a small problem. The root resides on a storage system device that we will call the _root physical volume_ (RPV). In order to validate the root we must validate the contents of the RPV. But validation of storage system devices is normally a file system function, which is what we are trying to initialize, so we have a problem. The solution is to regard validation of the RPV as a file system initialization function. This approach, for instance, has been taken in Multics. This solution is unpleasant since the RPV validator probably duplicates some of the regular device validator. But if the file system layer is unstructured, as we assumed, this is our only solution.

If the file system layer has more structure than we have assumed then it may be possible to use some of the normal verification machinery to verify the RPV. This would be possible if the file system layer actually consisted of two layers; an inner layer that knows about the physical implementation of the file system and an outer layer that knows the logical structure. If, in such a system, the inner layer has the validator built into it, then it may be possible to get the inner layer running without verifying the RPV and then use it to verify the RPV in the normal way.

6.6 Wrapup

In this chapter we have examined what is needed to initialize the file system layer. We have seen how the file system initializer needs dynamic reconfigurations in order to perform its task. A list of needed reconfigurations has also been presented. We have seen that the file system initializer must be divided into two parts: one loaded as part of the base layer core image and another, larger part, loaded by the first part. Finally a discussion of the addition of storage system devices was given as well as the validation of the root of the file system.

Chapter Seven

Dynamic Reconfiguration

In the previous chapters we have described a system initialization scheme that maintains the simplicity of the core image approach to initialization but, at the same time, remains as versatile as incremental techniques. The scheme presented depended, in large measure, upon the ability to perform a wide variety of dynamic reconfigurations. In this chapter we will discuss some of the engineering issues that one encounters in trying to implement these many and varied reconfigurations. Our discussion will be on the implementation of dynamic reconfigurations in the general context of a running system and not just for initialization time reconfigurations. We do this since at the time initialization invokes reconfigurations of a lower layer, the lower layer is running normally and it is not initialization time from his point of view. This is of interest because most operating systems provide few, if any, dynamic reconfigurations. It is not the purpose of this chapter to provide final answers to these issues but rather to point out the important issues and indicate some possible solutions.

7.1 Hardware Reconfigurations

Hardware reconfigurations fall into four categories - add a CPU, add memory pages or memory modules, add an input-output multiplexor (IOM) and add an I/O device. In his thesis, Schell discussed at great length the problems of adding memory and CPU's so we will not discuss them further here. We shall, instead, concentrate upon the addition of IOMs and I/O devices.

7.1.1 IOM Addition

The addition of an IOM is a four step procedure the effect of which is to open a communication path between the IOM and the operating system and to allow the addition of I/O devices physically attached to the IOM.

The first step in IOM addition is to place the IOM in a known, valid state. First the IOM is placed in an initialized state where it is performing no activities and is waiting for commands from a processor. In particular it should not be performing any I/O or attempting to send interrupts. This is either done by having the operator press an INITIALIZE button on the IOM (the approach taken in most present day architectures towards initializing a module) or by sending an INITIALIZE signal to it. Next we must ensure that the IOM is using the correct physical address space - we must ensure that the physical addresses that the IOM believes correspond to individual memory modules are correct. Finally we must validate its mailbox (1) address. The effect of these is that when communication between the IOM and processors is begun the IOM will not inadvertently overwrite parts of core with ongoing I/O and communications can proceed through a known area in core.

Next we must clear any pending interrupts associated with this IOM. (2) Since the operating system is not currently communicating with this IOM it has masked out the occurrence of these interrupts. When we open the communication path to the IOM we will be allowing interrupts on these cells to occur. Any currently pending interrupts on these cells will then cause spurious interrupts that the operating system may not handle in a safe manner. So for safety's sake we now clear these pending interrupts. The manner in which this is done is dependent upon the available hardware features. Ideally the processor has an instruction that allows the operating system to clear a pending

---

(1) The mailbox is an area in core used as the primary communication means between the IOM and processors.

(2) Each IOM has one, or more, interrupt cells assigned to it on which it generates interrupts to processors.

interrupt on some cell. If such an instruction is not available it is then necessary to:

i. Set up an interrupt handler for these cells that ignores interrupts on them.

ii. Set the interrupt mask so as to allow interrupts only on these cells, saving the old interrupt mask.

iii. Wait a short time to allow any pending interrupts on these cells to occur and ignore them when they occur (via the handler set up in i.).

iv. Set the interrupt mask to the saved one.

At the conclusion of this sequence no interrupts will be pending on these cells and we can safely proceed to the next step of adding the IOM.

The next step is to allow interrupts to occur on the interrupt cells associated with the IOM. This involves setting up the interrupt handler for these interrupts, which is probably the IOM manager. (1) He must be told that interrupts might be coming from this IOM, but that if any are received they should be ignored since there are no I/O devices on the IOM yet (as far as the operating system is concerned). Having informed the IOM manager about this situation, the system's interrupt masks are changed to allow interrupts from the new IOM.

The last step is to actually open the communication path between the IOM and the rest of the system. First we initialize the IOM's mailbox to some known state. Next we tell the memories to allow the IOM

(1) The routines that control the IOM: the manager of the IOM resource.

to access memory (up to now the IOM has been denied access since it was not part of the configuration in use). With this the complete physical communication path between processors, memory and the IOM is in operation. Lastly we inform the IOM manager that the IOM is ready so that he can use it fully. The IOM has been added.

This addition of an IOM was straightforward but it has one important implication - interrupt masks must be protected objects. By this we mean that it must be possible to verify that only valid interrupt masks (1) are used by the system. This is necessary to avoid problems before the third step above - allowing interrupts from the IOM to occur. Prior to the addition of this IOM, interrupts on the cells associated with it cannot be allowed, since the IOM manager, the normal handler of IOM interrupts, does not know about this IOM. (2) This means that all interrupt masks used by the system must be validated to only allow interrupts on "safe" cells. After step three it is necessary that those interrupt masks used allow interrupts from this IOM. This is so the IOM can interrupt the processor when it needs to. In particular we want to be sure that when the system wishes to allow IOM interrupts, interrupts are allowed from the IOM being added.

(1) For a fixed configuration there is a collection of interrupt masks that the system will use. At any instant in time in a situation where the configuration is changing due to dynamic reconfigurations, we want to be sure that the system is using interrupts masks corresponding to the current configuration and not to a previous one.

(2) An alternative would be to direct these interrupts to a handler that ignores them or to have the IOM manager ignore them since they come from an unconfigured IOM.

Assume there are a fixed number of masks that the system uses. There are two very similar ways to bring about this protection. First we could have an interrupt mask manager. Each time it is desired to use a different interrupt mask he is called to do it and told, "Change to mask i". (1) He does this by finding, or constructing, interrupt mask i and then causing it to be the interrupt mask in use. In this case to allow interrupts from the new IOM we need only inform the interrupt mask manager to open up the masks and he will do so. The other approach is to have the collection of interrupt masks currently usable by the system in one, fixed location. When it is desired to use a particular mask, one picks up the mask and then begins to use it. This operation of putting a new mask to use must be atomic so that the interrupt mask cannot change between picking up and setting the mask. These two methods are equivalent in that there is only one copy of any particular mask in the system. The first method simply provides centralized enforcement of masking conventions. Both methods achieve protection by only allowing interrupt masks to be used in atomic masking actions and only to be referred to by name elsewhere. In either method to perform step three one need only change the few, fixed masks and then ask each processor to remask (so as to ensure that they get the new masks).

(1) Where "i" is the name of the mask which one wants to use.

7.1.2 I/O Device Addition

The addition of an I/O device is simple once the IOM it is attached to has been added. First the IOM manager must be informed that I/O is now possible with this device. Second the manager of this particular (kind of) device must be informed of the device's existence so that the manager may update its local databases and, possibly, place the device in some particular state. The only possible difficulty in doing this is the updating of the device manager's databases as this may involve some of the problems mentioned below.

7.2 Software Reconfiguration

In this section we discuss some of the problems that may arise in performing software reconfigurations. We deal in general terms so as to avoid implementation details and to make our comments as widely applicable as possible. We deal with two types of software reconfiguration - setting of parameters and growing of system tables (databases).

7.2.1 Parameters

One class of dynamic reconfiguration is the setting of system wide parameters, such as scheduling parameters, and switches, such as metering on/off. Here we can give no guidelines across the whole spectrum of possible parameter reconfigurations, but we will discuss the two major kinds - metering and scheduling.

We can turn the metering of some function on or off as a reconfiguration. Internally this is relatively simple, since when we turn metering on we simply begin gathering statistics and when we turn it off we simply stop gathering statistics. The real issue is how users of the metering statistics being gathered are affected. This is a user interface problem and questions must be answered such as:

> If we turn off metering what do we do with the statistics we have already gathered?
>
> When we resume metering do we begin counting from where we left off or do we start from scratch?
>
> Do we need to inform the users of the meters that they have been turned on (off)?

In answering these questions one will formulate the high level mechanism of turning metering on/off.

Changing system wide scheduling parameters has its major effect internally; the effect on users is simply a change in performance, which is the intent. When we change system wide scheduling parameters the basis upon which we scheduled all current processes is invalid. We could, at this point, reschedule all processes based upon the new criterion. This is a workable solution but it may not scale well for if the number of processes is large or if the cost of rescheduling a process is high, for then the cost of rescheduling all processes may be prohibitive. An alternative is to apply the new scheduling criterion to individual processes as they come up for rescheduling. In taking this approach care must be taken to ensure that no processes are indefinitely kept in a state where they will not be scheduled to run.

7.2.2 Table Expansion

We have seen a number of cases where it is desirable to be able to expand a database dynamically - the core map, the active process table and the active segment table. In this section we discuss a number of issues related to such expansions.

In some cases it may be possible to avoid ever having to expand a table. Instead, at system generation time, it is allocated at its maximum possible size. Now when we need a new entry in this table it is already available and we can just begin to use it. Initial allocation at maximum size is only practical for tables of small maximum size, since allocation at maximum size could tie up excessive memory resources. For instance on Multics the number of memory modules is constrained by the hardware to be always less than or equal to eight so that the table describing memory modules can be initially allocated with eight entries. On the other hand the core map can have thousands of entries and thus be very large and so cannot be initially allocated at maximum size.

7.2.2.1 Supervisor Segment Growth

In attempting to grow a table it is, generally, necessary to grow the segment that the table resides in. We now consider some of the issues associated with growing these supervisor segments. The following discussion will consider such segment growth given that we are trying to grow a table that is in the segment.

These tables have two properties of major interest. First, they are supervisor segments and a process cannot lose its ability to address them. Second, these tables are generally core resident so that the space they use must be kept small (since memory is a scarce resource).

The first question is whether the table should reside in an unpaged segment (i.e. an N-word segment is allocated N contiguous words of memory) or a paged segment (i.e. it is referenced through a page table). An unpaged segment has the advantage that it only takes as much memory as is actually needed where a paged segment takes an integral number of pages (potentially wasting space on the last page). The effect is that the paged segment experiences space fragmentation within it. The unpaged segment may cause fragmentation of the physically available memory if, when we grow it, we must move it and thus leave a hole in

```
 _______________________________
|       |     |        |        |
|  In   |     |  In    |  Free  |
|       |  A  |        |        |     A= Unpaged segment
|  Use  |     |  Use   |        |          to be moved.
|       |     |        |        |
|_______|_____|________|________|

            Before Move


 ____________________________________________
|       |     |        |          |          |
|  In   |  F  |  In    |          |          |
|       |  r  |        |     A    |   Free   |
|  Use  |  e  |  Use   |          |          |
|       |  e  |        |          |          |
|_______|_____|________|__________|__________|

            After Move
```

Figure 7.1

memory behind it (see figure 7.1). To make efficient use of core using unpaged segments the system must be willing to reclaim these holes. In a paged virtual memory system, such as Multics, such a reclaimer is unlikely to exist and to build one for this special purpose seems to be a waste of effort. For this reason keeping tables in unpaged segments is, in general, less desirable than maintaining them in paged segments. The one exception is when the maximum table size is small and known, where initially allocating an unpaged segment of the maximally needed size is reasonable. In short, any table that might be grown should be kept in a paged segment.

Another problem is that there are many processes referencing these tables and none can lose addressability since they are in supervisor segments. This means that care must be taken as we grow the segment. If we grow an unpaged segment, we must move the data to the new location and update _all_ descriptor segments that describe the segment; this must be an atomic operation with respect to references to this segment in _all_ processes. This may be very difficult to do since the number of processes may be large and the database may require frequent accessing. This is another argument against using unpaged segments for growable tables.

For paged segments this is not as serious a problem. To grow a segment we need only allocate a new page, fill in the segment's page table to reflect the new page then go around and change all SDW's to reflect the new length. Only now, after all this is done, need we lock the database to add the new entries. This differs from the unpaged case where we had to lock the database while we were changing all SDW's (and this is the time consuming part) in order to prevent access to the table.

7.2.2.2 Multiple Tables in a Segment

The next set of problems we will discuss occur as a result of having more than one growable table in a segment. This packing technique is used in Multics, for instance, to reduce breakage in supervisor segments and reduce the number of segments in the supervisor.

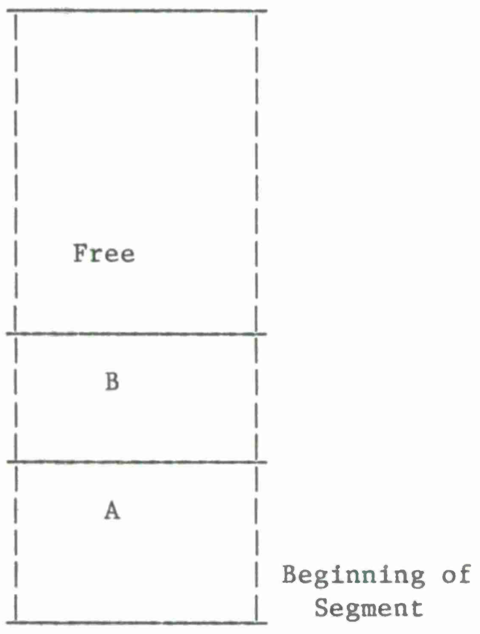

Figure 7.2

For instance in figure 7.2 suppose that A and B represent tables that both can be grown.

If we wish to grow B we simply grow the segment, if needed, and allocate the new entries for B. No problems occur.

Now consider the case of growing A. In order to grow A it is necessary to make room for a larger A within this segment. This can be done in one of two ways: move A so it follows B in the segment or move B towards the end of the segment and grow A in place. If we move A to the

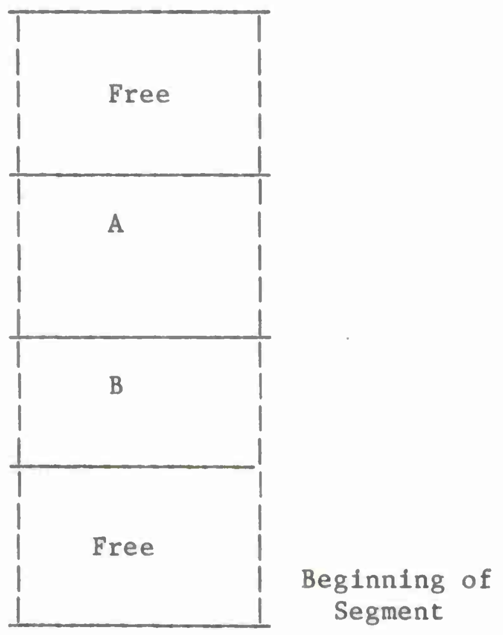

Figure 7.3

end of the segment we then have the situation in figure 7.3 The disadvantage of this scheme is that we create "holes" of unused space in the segment as we reconfigure. These holes may represent wasted memory resources and so are undesirable. In any event we must keep track of them so that we may use them later if needed. The alternative, moving B, does not create holes but it does have the effect that a reconfiguration of A causes B to be moved. If an undetected error should occur while moving B we could have the unpleasant situation where a reconfiguration of A has caused a function associated with B to fail.

In both cases above we have had to move a table. We now must consider some of the factors that can make the moving of a table difficult.

The first problem is one of locking. When we move a table we run into the problem of someone updating the table during the move. (This problem is analogous to that noted by Schell in his thesis while copying pages out of a memory that is being deleted). To prevent someone from performing such modification we must be able to set a lock to prevent access to the table. If access to the table is controlled by a global lock we need only set it, move the table and unlock. On the other hand if some system of finer locks is used to control access to the table, we will need some means of getting possession of all of these locks. One way of doing this is by having a global lock that, when set, prevents the setting of any of the finer locks. To move the table we then set the global lock, wait for any finer locks to be unlocked, (1) move the table and then release the global lock. In either of these cases we can move the table with the assurance that no modifications will take place

(1) We are assuming that no lock will ever be set for an indefinite time.

during the move since all access is locked out. It may also be necessary to lock out reads of the table during the move so that after we make the copy the actual table, no one will still be accessing the old table (and hence out-of-date information).

Another problem in moving a table concerns saved pointers to a table. When we move the table we must be able to change all pointers to (or into) the table to reflect its new location. Failure to do so will leave some pointer pointing to the old table location which, now, does not have valid table data. This can be solved with a combination of two techniques. First there should be _one_ pointer to the base of the table kept in a fixed, known location. All accesses to the table must be through that pointer. Also that pointer must be protected by the global table lock (1) so that when the table is moved only correct pointers to the table are used. Second, pointers into the table must either be offsets relative to the base of the table or they must be indexes of table entries. In either case a pointer to the actual data in the table can only be generated by use of the _one_ pointer to the table base and this relative offset.

Another problem limiting the ability to move tables is the presence of immovable data. In the last paragraph we described how virtual memory pointers could make a table hard to move. Tables can also be

(1) If a finer system of locks is being used, the setting of any finer locks should prevent modification of this pointer.

hard to move due to absolute pointers (1) to the table. Examples include page tables (used by the hardware memory mapping algorithms) and I/O databases that the IOM, in most present day systems, has absolute pointers to. Note that almost all such pointers are for the use of the hardware. The number of such pointers can be rather large and may be difficult to update. For this reason databases pointed to by absolute pointers are probably immovable unless some major redesign of the hardware is undertaken to minimize the number of absolute addresses in the system and make such addresses easily modified. Such a design is, however, beyond the scope of this thesis and so we must consider tables pointed to by absolute pointers to be immovable.

The issues above have been related to moving one of several tables in a segment in response to a request to grow one of them. This problem can be avoided entirely if all of the tables in the segment are maintained solely by linked lists. In this case to grow a table we need only get some space at the current free end of the segment, create new entries there and link them into the relevant table. There is no need to move any current data.

The best answer to all of these questions regarding moving tables is to never have to move them. We can easily arrange this by having at most one growable table per segment.

(1) An absolute pointer is one that points to a particular real, not virtual, address.

7.3 Multics

As part of the research for this thesis Multics was examined to see how difficult it would be to add additional dynamic reconfigurations. To this end we investigated the dynamic addition of IOM's, and the reconfiguration of the active segment table (AST), which holds page tables for active segments, was designed and successfully implemented. This second facility allows one to dynamically create AST entries.

The ability to dynamically add an IOM was investigated as part of this thesis. This investigation resulted in the paper design presented earlier in this chapter. Two items of interest resulted from this investigation. First the actual addition of the IOM is not really very hard. Some minor problems, and in particular the protection of interrupt masks, must be overcome but no major ones. Second, it was found that the actual reconfiguration code could be taken, in large measure, from the current code that initializes the IOM related databases. This was a surprising result which, if true more generally, would make retrofitting dynamic reconfigurations onto Multics fairly easy.

In developing the ability to dynamically reconfigure the AST no major problems were encountered but two minor problems, not previously considered in this chapter, were encountered. First was the fact that

that the data in the table, the page tables, was directly referenced by the hardware. This meant that care had to be taken to ensure that the page tables were laid out properly. In particular, the page tables must be contiguous in physical memory (as opposed to simply being contiguous within the segment) so that the hardware memory mapping algorithm works properly. This was a problem since the segment containing the AST was made a paged segment as part of implementing this facility and, as such, page i+1 of the segment might not be immediately after page i in physical memory. In particular a page table that is split over a virtual page boundary must still be contiguous in physical memory. The solution was simply to make the reconfiguration program aware of the physical addresses of the pages in the segment and act accordingly in creating new AST entries.

The second problem was software conversion between physical and virtual addresses. It turns out that frequently the Multics supervisor needs to translate the virtual address of some object to its physical address and also from a physical address to the corresponding virtual address. This is needed, for instance, to find a page table in virtual memory given its physical address (this occurs while handling page faults) and to find its physical address given its virtual address (this occurs when trying to fill in a segment descriptor word while processing a segment fault). In the current version of Multics this translation is very easy, a simple addition or subtraction involving the absolute address of the segment and an offset within it, since the segment containing the AST is known to be contiguous in physical memory. When the reconfiguration of the AST was implemented it was no longer possible to guarantee contiguity of the AST in physical memory since the segment containing the AST was made a paged segment. The solution was to modify this translation code to take into account the lack of contiguity in the segment.

The actual details of how these problems were solved are not important. The important thing is that in implementing dynamic reconfigurations one is going to encounter minor problems that will have to be solved. In examining the various kinds of reconfigurations needed on Multics, and how to implement them, no fundamental problems were encountered. We conjecture that no fundamental problems would be encountered in trying to provide dynamic reconfigurations on most other systems.

7.4 Wrapup

In this chapter we have discussed some of the issues associated with dynamic reconfiguration. The addition of processors and memories was covered in Schell's thesis. We covered the addition of IOM's in detail. The important result was that interrupt masks must become protected objects. Addition of I/O devices was then seen to be trivial once the problem of IOM addition was solved. Software reconfigurations were then discussed in terms of parameter changing and table expansion. The discussion of parameter changing concentrated on two cases, turning metering on/off and changing scheduling parameters. A number of issues were discussed with regards to table expansion. The conclusion here is that, to best facilitate table expansion, in general there should be at most one variable-size table in a segment and that segment should be a paged segment (as opposed to an unpaged segment).

Chapter Eight

Conclusion

In this thesis we have explored system initialization, the problem of bringing an operating system up on a particular machine. The major result of this thesis is a method of initializing systems that is both simple and easy to understand and which, at the same time, has the property of being versatile in the face of configuration changes. Most current systems employ a method that is very simple, a core image approach, but which cannot easily handle configuration changes. Multics, on the other hand, uses a method that is versatile but which also is rather ad-hoc and difficult to understand. The method presented in this thesis maintains the simplicity of the core image approach and the versatility of the Multics method.

8.1 Results

We have considered initialization in the context of a layered system. In such a system, initialization proceeds upward, layer by layer, first getting layer i initialized and running and then running upon it to get layer i+1 initialized; eventually the whole system is initialized. In this layer by layer approach the hardest part is initializing the base layer of the system.

The thesis presents a method of initializing the base layer which essentially reduces its initialization to the loading of a core image of the base layer. This is done by hypothesizing the existence of a minimal configuration - a configuration that is a subset of all possible viable configurations. Such a configuration does seem to exist for the centralized architectures typical of present day general purpose computer systems. We now assume that the system will be running on the minimal configuration, allowing us to create, at system generation time, a core image of the base layer as it should appear in core. We can only create this core image once we have assumed some configuration. The minimal configuration is a useful one to choose since a system that can run on the minimal configuration can run on any viable configuration.

The minimal configuration consists of one processor and some memory. We assume the existence of both a certain amount of memory as well as the existence of physical addresses in memory.

When it is desired to actually get the system up and running one must first get the base layer running. Since we have created a core image of it this simply involves loading the core image into memory, validating the correctness of the load operation, and then giving the base layer control. At this point the base layer must ascertain the system wiring diagram corresponding to the minimal hardware configuration. This is necessary since we did not assume this knowledge as part of the minimal configuration for to do so would reduce the number of configurations the base layer could run on (i.e. the assumed configuration would no longer be minimal). After ascertaining this knowledge the base layer will be running after having only performed a load operation and a small amount of processing.

Once the base layer is operable, control is given to the file system initializer. It is here that the second key concept, dynamic reconfiguration, comes into play. In the process of initializing the second layer of the system, the file system initializer will invoke dynamic reconfigurations of the base layer to cause it to run on the configuration actually present. Note that it is only the availability of these dynamic configurations that makes the assumption of a minimal configuration a workable one, as otherwise the system would always run on the minimal configuration.

File system initialization consists of invoking dynamic reconfigurations of the base layer to gain access to the bootload medium, an initial paging area and the secondary memory that the file system resides on. Access is needed to the bootload medium so that the components of the file system layer may be be obtained since they are, by convention, on the bootload medium. An initial paging area is needed since the file system layer is, presumably, too large to fit in core all at once so that an operable virtual memory mechanism is needed in order to get it running. Note that up to this point the paging mechanism is working but is not too useful since it does not have any secondary storage to use as a paging pool. Before telling page control to use an area, the file system initializer must verify that the paging area can be used. This consideration, plus the possibility of a previous system failure, led us to conclude that a separate area on secondary storage must be reserved for use as the initial paging pool. The file system initializer must gain access to the secondary storage that the file system is on so as to perform validation of its contents since a previous crash may have left the file system in an inconsistent state.

The remaining tasks of file system initialization are straightforward. The programs that comprise the file system layer must be loaded into the system's virtual memory and pre-linked. Also the databases used by the file system layer must be set up. This completes file system initialization and control is passed outward to the next layer in the system.

We concluded the thesis with a discussion of implementing dynamic reconfigurations. The addition of an input-output multiplexor and I/O devices were discussed as hardware reconfigurations. The changing of software parameters was briefly discussed. The expansion of software tables was covered in detail. None of these needed dynamic reconfigurations was found to present major, or fundamental, implementation problems.

8.2 Tradeoffs

In considering the approach proposed in this thesis the tradeoff between initialization time complexity and system run time complexity must be appreciated.

Fundamentally, the scheme proposed here depends upon the existence of many dynamic reconfigurations. The introduction of these dynamic reconfigurations to a system that does not already have them increases, even if only slightly, the complexity of the system. If they are not otherwise needed this is a negative factor. On the other hand the existence of these dynamic reconfigurations increases the flexibility of the system and reduces initialization complexity a great deal. The system designer must weigh these two factors and decide at which end, or where in-between, he wants his system to lie. In helping the designer make this decision we offer our opinion based upon our experience with Multics. The presence of the dynamic reconfigurations increases system flexibility a great deal and reduces initialization complexity a great deal, all at a rather modest increase in system complexity. We also feel that the system as a whole, including initialization, will be more easily certified as correct with these dynamic reconfigurations than without because of the simplicity gained at initialization time.

8.3 Further Research

The results of this thesis suggest five areas where further investigation is warranted. Four are concerned with dynamic reconfigurations. First an investigation of the tradeoffs involved in providing dynamic reconfigurations should be undertaken. In this thesis we have seen how allowing system segments to be growable can result in space wastage. Is this inherent? Can other approaches eliminate it? Do other, unexamined, dynamic reconfigurations incur space/time wastage? Second is the issue of DELETE type reconfigurations. In this thesis we found need for many ADD type reconfigurations. In a computer utility one would also want the complementary DELETE type reconfigurations. An investigation of the engineering issues involved in providing these is needed. Underlying the whole area of dynamic reconfiguration is a third area of possible future research - hardware. In his thesis, Schell found that the dynamic reconfiguration of processors and memory was facilitated by certain features of the underlying hardware. Could hardware features be found to facilitate the dynamic reconfigurations mentioned in this thesis? For the complementary DELETE type reconfigurations? The fourth area of future research concerns the formal specification of dynamically reconfigurable systems. Current papers on formal system specification, such as [SRI], do not discuss dynamic reconfigurations. Examining such a design, it is not obvious, to this writer, how to specify dynamic reconfigurations. Research into the formal specification of dynamic reconfigurations is clearly needed. The fifth, and last, area where future research might be directed is towards the general use of contiguous physical memory allocation within the supervisor. Such allocation might be efficient for things such as IOM mailboxes and I/O buffers. For general use in the system (i.e. for user programs and data) the problems of periodic compaction of memory may be difficult to deal with efficiently. In the limited context of the supervisor, where the situation is fairly static and the allocations come in only a few sizes, is the problem more manageable? Can contiguous allocation be put to use in an efficient manner?

BIBLIOGRAPHY

[CDC] *Scope 2.1 Installation Handbook*, Control Data Corporation, 1974.

[Dijkstra] Dijkstra, E.W., "The Structure of the 'THE' Multiprogramming System", *CACM 11*, 5 (May 1968), pp 341-346.

[Flores] Flores, Ivan, *Operating System for Multiprogramming with a Variable Number of Tasks*, Allyn and Bacon, Inc., Boston, 1973.

[GM] Ward, M.R., "The GM Multiple Console Time Sharing System. A Simple Approach to Operating System Generation and Initialization", *SIGOPS 10*, 1 (January 1976), pp. 61-70.

[HISIa] *System Initialization Program Logic Manual*, Honeywell Information Systems Inc., Order number AN70, 1975.

[HISIb] *Multics Processor Manual*, Honeywell information Systems Inc., Order number AL39, 1976.

[Huber] Huber, A., "A Multiprocess Design of a Paging System", M.I.T. Laboratory for Computer Science Technical Report 171, 1977.

[IBM] *Catalog of Programs for IBM System/360, Models 25 and above*, IBM Corp., Order number GC20-1619-8, 1970.

[MAC] "Introduction to Multics", M.I.T. Project MAC Technical Report 123, 1973.

[MSPM] *Multics System Programmers' Manual*, M.I.T. Project MAC, 1967.

[Reed] Reed, David P., "Processor Multiplexing in a Layered Operating System", M.I.T. Laboratory for Computer Science Technical Report 164, 1976.

[Schell] Schell, Roger R., "Dynamic Reconfiguration in a Modular Computer System", M.I.T. Project MAC Technical Report 86, 1971.

[SRI] Neumann, P.G., et al., "A Provably Secure Operating System", Final Report of SRI Project 2581, Stanford Research Institute, Menlo Park, Calif., 1975.

[Stern] Stern, Jerry A., "Backup and Recovery of On-Line Information in a Computer Utility", M.I.T. Project MAC Technical Report 116, 1974.