TS 1015

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-180

A SIMPLE AND FLEXIBLE SYSTEM INITIALIZATION MECHANISM

Allen W. Luniewski

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

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

ACKNOWLEDGMENTS 2

ABSTRACT 3

TABLE OF CONTENTS 4

LIST OF FIGURES 6

1. Introduction 7

1.1 Initialization in General 8
1.2 The Need for Versatility 10
1.3 Related Work 14
1.4 Thesis Outline: Preview of Approach 15

2. A Model of a Computer System 19

2.1 Hardware Base 19
2.2 Software Base 21
2.3 Multics Initialization 23
2.4 Wrapup 27

3. Overview of the Initialization Scheme 29

3.1 Initialization in a Layered System 29
3.2 Base Layer Initialization 32
3.3 Wrapup 36

4. Core Image Generation 37

4.1 The Process of System Generation 37
4.2 Where System Generation Occurs 39
4.3 The Minimal Configuration 40
4.3.1 Main Memory Size 41
4.3.2 Main Memory Addresses 43
4.3.2.1 A Relocating Loader 44
4.3.2.2 Realization of Assumed Addresses 46
4.3.3 System Table Sizes 49
4.4 Common Actions 50
4.5 Wrapup 53

5. Base Layer Loading and Initialization 55

5.1 The Core Image Loader 55
5.2 Core Image Size 58
5.3 Base Layer Initialization 60
5.4 Wrapup 62

6. File System Initialization 63

6.1 Dynamic Reconfiguration 63
6.2 Reconfiguration 64
6.3 File System Initialization 66
6.4 The Initial Paging Area 69
6.5 The Root 72
6.6 Wrapup 73

7. Dynamic Reconfiguration 75

7.1 Hardware Reconfigurations 76
7.1.1 IOM Addition 76
7.1.2 I/O Device Addition 81
7.2 Software Reconfiguration 81
7.2.1 Parameters 82
7.2.2 Table Expansion 83
7.2.2.1 Supervisor Segment Growth 84
7.2.2.2 Multiple Tables in a Segment 87
7.3 Multics 92
7.4 Wrapup 95

8. Conclusion 97

8.1 Results 98
8.2 Tradeoffs 101
8.3 Further Research 102

BIBLIOGRAPHY 104

LIST OF FIGURES

Figure 7.1 .....
Figure 7.2 .....
Figure 7.3 .....

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 the methods 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 one 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 problem of the Multics initialization scheme.

for the I/O to complete. It then changes 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