IA-32 Operating System

James Mackenzie

March 2021

Abstract

This paper follows a linear path of breaking down the core concepts of the IA-32 architecture with the defined goal of creating a portable operating system capable of running arbitrary binaries. First begining with an overview of different operating system architectures and the IA-32 architecture itself, this paper discusses key factors, such as the features and functionality of the architecture. Followed by a breakdown of the design approach necessary to achieve the operating system. The paper continues by explaining how an IA-32 system can be utilised to build a monolithic operating system in C and assembly using the GNU GRUB bootloader to do the heavy lifting. This is followed by an evalutation of the ability of the kernel to execute certain binaries. This paper concludes with the ability to execute binaries, on a light weight, portable operating system.

Contents

List of Figures

Chapter 1

Introduction

Before operating systems, a program would have to be developed to run directly on hardware. This meant that a program had to be tailored to only one specific machine. Because of this, the program was denied any sort of portability.

Operating systems ecouraged mass adoption by promising the ability to run multiple arbitrary programs in a secure environment, without knowing or having to accomidate the hardware specifications of the machine. This was done using abstraction and allowed for developers to create programs that were non-hardware specific, meaning that they could be run on any machine that was running said operating system. Because of this, software was free to flouish and allowed for the mass software expansion that we now see today. These operating systems provided various features, one of which was the guarantee of security. This meant that untrusted programs could be run on a machine with the guarantee that it would not interfere with any other programs or harm the machine in any way.

The overall goal of this project is to provide a lightweight, portable monolithic operating system that can run arbitrary programs from a file system reliably and at a fast speed, while still providing the same features and security guarantees. This means that the operating system will employ memory protection, usermode, system calls and interrupts. On top of this, most monderndav operating systems prefer to be installed to the disk before they can be booted. This operating system is to be portable, meaning that it needs no installation to run and can simply be booted from a disk file.

The many choices made to provide this operating system solution is discussed in the following chapter.

1.1 Aims

1. Explore the IA-32 architecture and provide a write up of the major features it offers.

2. Compare and contrast different algorithms and implementation methods to provide a fast operating system.

3. Provide a written guide on how to use the IA-32 architecture to set up a basic operating system.

4. Explore the ELF file format to provide the ability to run arbitrary programs in a secure environment.

5. Provide the same security guarantees that are expected from a commercial operating system.

1.2 Topic Coverage

- **Chapter 2**, Background, sets up a basic view of the architecture of the project in general. It explores the different operating system architectures, the IA-32 system architecture, and a description of the tools that will be used to realise the project.

- **Chapter 3**, Design, takes a tour through the design decisions that were made to achieve the project outcome. It describes the implementation choices of certain features and algorithms in tone with the goals of the project.

- **Chapter 4**, Implementation, touches upon the various implementation methods used within the project itself. It walks through the step by step procedure of implementing a monolithic operating system on the IA-32 architecture.

- **Chapter 5**, Testing, examines the finished operating system and determines if the goals and aims of the project have been met.

- **Chapter 6**, Conclusion, reviews the outcome of the project, why and why not certain goals and aims were and were not met, any planned future work and some final remarks.

Chapter 2

Background

2.1 Operating Systems

An operating system in simple terms is a piece of software that runs on a computer system to provide a platform to run various other pieces of software. It strives to provide a view of a computer system that is more easier to understand to the average programmer by using various abstraction methods. An operating system is stocked with an abundance of software libraries, drivers and modules to make it simple for a programmer to interact with the hardware without having to write their own code to do so; this would take a long time and deny any sort of portability. On top of this, the operating system manages various different components of a computer system, such as keyboard input, memory, the display, the disk and more.[1]

With the operating system serving as a system resource manager, programs are free to do their own thing. An operating system can handle multiple different programs without them interfering with each other in any way. Programs have their own environment in which they execute, meaning that they get their own memory, registers and stack[2]. This environment cannot be seen by other processes unless the operating system is directly queried by a process looking to do so. This environment provides a layer of security to processes running on the operating system while also ensuring that a programmer does not have to account for other processes using system resources.

The basic architecture of an operating system consists of a kernel and user mode processes. Kernel mode is the highest priviliaged state of the processor and is where the kernel lies, as the kernel must have unlimited access to the entire system for management purposes. User mode processes run at a lower priviliage (commonly the least priviliaged state) than the kernel, with a subset of the instructions that they can perfrom, and will be discussed in more detail

[1]More in-depth information about operating systems can be found in Modern Operating Systems (3rd Edition) by Andrew S. Tanenbaum.[1]

[2]A stack is a data structure stored in memory that is used for passing arguments to functions and storing the return address.

later (see 2.2.3 Ring Priviliage Levels, page 8). For now, it is to be assumed that they exist in this lower priviliage state to ensure that they cannot do too much damage to a system.

If a program requires a resource from the kernel, or wishes to perform an elevated action, it will contact the kernel via a method known as system calling. System calling is the process of a program calling a kernel function from user space and will be discussed in more detail in the later sections (see 2.2.8 System Calling, page 14).

There are two main types of operating system architectures, monolithic and microkernel. Each providing their own advantages and disadvantages. They are discussed in the following sections.

2.1.1 Monolithic

The monolithic approach places all system services within a single binary process that runs in kernel mode. This single binary can call upon any function of the kernel and load drivers to perform more tasks and provide greater features. A driver is a program that is loaded by the kernel to handle communication with a hardware device; these also run in kernel mode. Because of this, the operating system can be assumed as being unreliable as loading a bad driver or triggering a bug in one of the system functions will cause a system crash. However, implementation of the kernel become much simplier as layers do not have to be built to expose certain functionaility to user mode processes. Nor does there need to be any inter-process communication to talk to servers and drivers (see 2.1.2 Microkernel, page 5). Instead, a more straightforward approach is taken. A set of system calls are exposed by the kernel, when a process needs to perform an elevated action it calls one of these system calls and the kernel handles everything from there (figure 2.1).

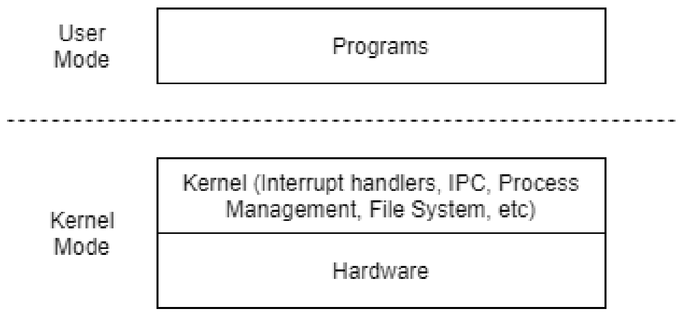

User Mode
Programs
Kernel Mode
Kernel (Interrupt handlers, IPC, Process Management, File System, etc)
Hardware

Figure 2.1: The monolithic architecture.

Software and hardware (or I/O) devices communicate with the kernel via interrupts. All interrupts, hardware and software based, are serviced by the kernel itself (usaully achived using drivers loaded by the kernel), as opposed to the microkernel which puts its drivers in user mode - this results in a slower handling process.

As the kernel is a single binary, memory management is achieved by the monolithic approach fully within kernel mode. This makes it a fast process, albeit sacrificing some system stability. However, is a must when considering the speed of the kernel itself. Discussed in the next section, explains how microkernels use a more stable, yet slower method of handling memory.

An example of a monolithic kernel is Linux. Linux is a widely used kernel that is highly regarded for its speed and feature set. It was adopted by major operating system projects such as Ubuntu because of these reasons.[3]

2.1.2 Microkernel

An argument has been made that the more code there is executing in kernel mode, the higher the probability of a bug being executed and crashing the entire system. The microkernel approach argues that the kernel should provide the most minimal functionaility as possible and leave the rest up to user mode servers and drivers. Servers being available to provide most of the kernel functionaility from user mode. These servers and drivers can be loaded, much like in the monolithic approach (see 2.1.1 Monolithic, page 4), however, will be executed from user mode. This ensures that if a bad server or driver is loaded or a bug is encountered, the system will not completely crash. This process involves creating a user mode process heirarchy using layers in which daemons (background processes) have different amounts of priviliage to handle different parts of the system. System calls are still exposed by the kernel, however, are limited in nature and are simply there for servers and drivers to perform elevated tasks and allow for inter-process communication; the act of processes communicating with each other, a must for a microkernel (figure 2.2).

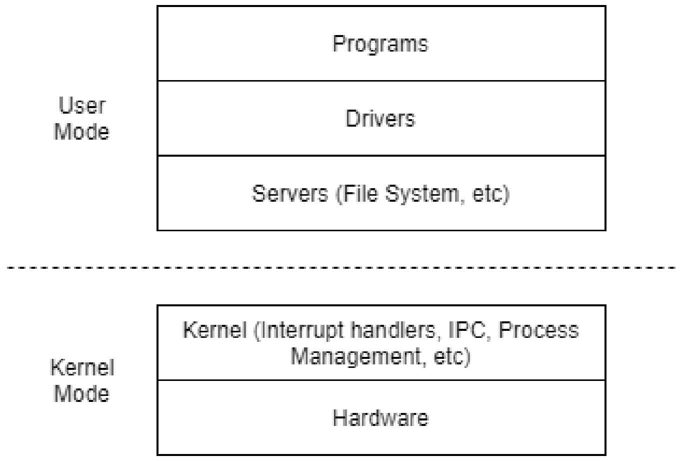

User Mode
Programs
Drivers
Servers (File System, etc)
Kernel Mode
Kernel (Interrupt handlers, IPC, Process Management, etc)
Hardware

Figure 2.2: The microkernel architecture.

[3]More information on Linux available at: https://www.linux.org/.

Message passing within a microkernel is commonly achived via message queues. This is the process of a server listening for messages to appear in a queue, and then reacting to the data that was put there by other processes. Because of this message passing, microkernel are said to be slower than their monlithic alternative, which uses system calls to communicate with hardware instead. System calls require no wait time, as they directly interrupt the execution of the central processing unit, whereas message passing has a slight delay until the process is scheduled to be run.

On top of this message passing dilemma, the microkernel memory management process must first travel through the kernel and into user mode. This means that if a page fault should occur a tedious process is undertaken, where the system moves from user mode to kernel mode, then back to user mode where the page server resides, back through kernel mode again and up into user mode to return to the original processes. Again, this is a time consuming task and is not optimal for a time integral operating system.[4]

A more adaptable approach is taken considering system drivers and servers. The microkernel does not handle software or I/O interrupts itself, its only involvement is ensuring the communication between a process and a driver/server. This means that the code of a driver/server can change without having to recompile the entire kernel. Not only this, but additional drivers and server can be loaded without having to make any changes to the kernel itself. This is major benefit for development of the system and ensures a smaller kernel size, assisting in portability. Due to the smaller size of the kernel architecture and the fact that its features are running as user mode processes, it is much easier to ensure the security and stability of the system which can be seen as a benefit compared to the monolithic approach.

This section is not saying that a microkernel does not have potential to be faster than a monolithic kernel. A microkernel can achieve fast speeds if tailored to the processor that it is built for. This becomes quite a cumbersome approach, however.[5]

2.2 IA-32 Architecture

According to the Intel developer manuals, the IA-32 architecture "consist of a set of registers, data structures, and instructions designed to support basic system-level operations such as memory management, interrupt exception handling, task management and control of multiple processors"[3].

Introduced in 1985, the 80386 processor was the first 32-bit processor for the IA-32 family of processors and provided the groundwork for the modernday IA-32 architecture. The datasheet describes it as an "advanced 32-bit micropro-

[4]However, this was corrected in L4 microkernels (second-generation microkernels) where a user mode system process is granted the rights to the entire memory. Processes can then call upon this process to receive memory without going through kernel mode.

[5]More information on the Monolithic and Microkernel architecture can be found in Monolithic kernel vs. Microkernel by Benjamin Roch.[2]

cessor optimized for multitasking operating systems and for applications needing very high performance"[5]. It provides 32-bit registers with data paths supporting 32-bit addresses and data types, while also having the ability to address up to 4 gigabytes of physical memory and 64 terabytes of virtual memory.

The instruction set introduced by this processor is still being used today and provides various 32-bit instructions to command the processor (see 2.2.1 Instruction Set, page 7).

The architecture provides a total of sixteen basic program execution registers, with an additonal set of model specific registers (MSRs), that can be manipulated via the vast instruction set provided by the IA-32 architecture (see 2.2.2 Registers, page 8).

The architecture also provides Ring Priviliage Levels (RPLs) which can be used to manage the abilities of an executing program and can be modified via the segment registers (see 2.2.3 Ring Priviliage Levels, page 8).

Furthermore, the architecture provides memory management through paging. This allows for the inclusion of virtual addresses that are treated as their physical counterpart (see 2.2.4 Virtual Memory, page 9).

To display data to users, a Video Graphics Array (VGA) buffer is used by the architecture to output text to the screen (see 2.2.5 Video Graphics Array, page 11).

Moreover, the architecture provides a Global Descriptor Table (GDT) and an Local Descriptor Table (LDT), that allows for implementation of segmentation, meaning that programs can be independant of the physical adress space in which they occupy, so that programs can be written without any sound knowledge of where they reside in memory and how much physical memory is available to them (see 2.2.6 Global Descriptor Table, page 12).

External interrupts, software interrupts and exceptions are dealt with using the interrupt descriptor table, which is a table of gate descriptors mapped to an interrupt number (see 2.2.7 Interrupts, page 13).

Finally, the architecture provides various assembly instructions in order to generate and handle system calls (see 2.2.8 System Calling, page 14).

The Intel developer manuals state that "Intel 64 and IA-32 architectures have been at the forefront of the computer revolution and is today the preferred computer architcture, as mesured by computers in use and the total computing power available in the word"[4]. Because of this it is safe to assume that the architecture is widely adopted and will continue to be supported well into the future. Knowing this, it is easy to see why this is an optimal architecture that meets the standards for a basic operating system to be deployed on.[6]

2.2.1 Instruction Set

The IA-32 architecture provides a 32-bit instruction set. This instruction set is known an an assembly language and can be assembled into machine code using

[6]More information about the IA-32 architecture can be found in the Intel 80386 Programmer's Reference Manual 1986.[5]

an assembler. Machine code is the binary instructions that are read and executed by a processor. Assembly languages provide a more readible instruction set that has one-to-one mapping with machine code.

The Intel assembly language provides the building blocks for creating software on the IA-32 architecture. Higher level languages like C are compiled to assembly before becoming machine code. However, it can sometimes be quite tedious to perform certain actions using a high level language like C (e.g. system calls (see 2.2.8 System Calling, page 14)), this is where assembly comes in. Using assembly, human readable code can be written to execute certain tasks directly on the central processing unit.

This architecture provides various different assembly instructions that will be mentioned in later sections.

Intel IA-32 assembly can be written in any text editing program and assembled using various compiler libraries such as GNU's GCC (see 2.5 C Programming Language, page 17).

2.2.2 Registers

A register is a fast memory store within the processor that can commonly store up to four bytes of data. The architecture provides a total of sixteen basic program execution registers that can be manipulated by the programmer. These are the general purpose registers (GPRs), the segment registers, the flags register and the instruction register. These registers are common throughout most modern architectures and most are generally used to store arbitrary data, however more detail around these registers will be added in later sections. The register names and types have been listed below:

- General Purpose Registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP)
- Segment Registers (CS, DS, SS, ES, FS, and GS)
- Flags Register (EFLAGS)
- Instruction Pointer (EIP)

On top of these registers, there are a few defined model specific registers (MSRs), that can be used to toggle certain features of the architecture on and off, these are accessed via the *wrmsr* and *rdmsr* assembly instructions when in kernel mode to write and read respectively. Some MSRs and their uses specific to this project will be mentioned in the later sections.

2.2.3 Ring Priviliage Levels

The IA-32 architecture employs the use of four different ring priviliage levels, also knowns as RPLs, each having different limits on the number of features that can be manipulated within the system. These RPLs are heirarchy based, meaning that the lowest RPL has the highest priviliage and the highest RPL has the lowest. The heirarchy can be thought of as an onion, with the outer layers being the least priviliaged and the inner layers being the most (figure 2.3).

The RPLs provide protection by restricting access to addressable domains, procedure entry points and instruction sets. These levels allow for protection agaist untrusted procedures making mistakes, or accessing data in which they should not. This is a powerful system that is made use of by many kernels today as it allows for untrustworthy software to run in a safe enviroment seperate from the kernel.

The RPL can be manipulated via the segment registers (see 2.2.2 Registers, page 8). By providing these segment registers with an offset into the GDT, cetain flags can be set to tell the processor which RPL it should currently be in.

Knowing this, it is easy to see how RPLs are an intergral part to building an operating system and as they provide the ground work for kernel mode and user mode, will be useful to this project.

It should also be noted before continuing that the terms *"user mode"* and *"kernel mode"* will be used interchangeably with the RPL value of 3 and 0 respectively.

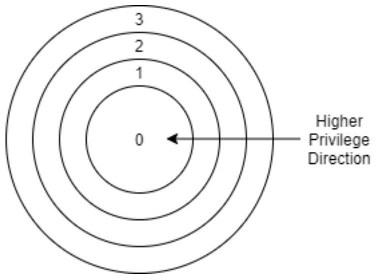


Figure 2.3: The RPL onion.

2.2.4 Virtual Memory

All computer systems must utilise a memory of some form. Memory is simply used to store data (e.g. programs, program data, files, etc) and can be read or written to by the processor. When the processor needs to execute the next instruction of a program, or wants some data that an instruction is requesting, it goes to an address in memory. Addresses are a four byte value that directly corrospond to a location in the system memory. It should be mentioned that sending and retreiving from memory is considerably slower than using the registers found within the processor (see 2.2.2 Registers, page 8).

The IA-32 architecture provides virtual memory. Virtual memory allows the processor to control what memory processes can and cannot access, while also allowing them to utilize memory addresses that are much larger than the address range supported by the memory, which is up to four gigabytes; this is achieved by *"paging"* (or moving) the least accessed memory regions to the disk and then moving them back into memory when they are required. Due to this system, virtual memory can address up to 64 terabytes of memory.

Virtual memory can be implemented using a memory structure known as the page directory. The IA-32 architecture provides two model specific registers (MSRs) that can be used to implement virtual memory using the paging system. The first being the cr0 MSR, that is used to set various control flags, and the second being the cr3 MSR, that is used to set the address of the page directory (figure 2.4). The Intel developer manuals describes bit 31 of the cr0 MSR being the paging bit, when this bit is set to 1 then virtual memory is enabled.

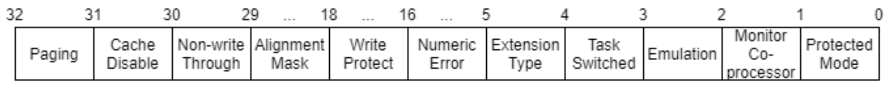


Figure 2.4: The various settings of the cr0 MSR.

The cr3 MSR contains the address of the page directory. The page directory is a table of 1024 four bytes entries, each pointing to a page table[7]. Each page table contains 1024 four bytes entries of the physical address that a 4 kilobyte page memory offset maps to. Virtual address can then be translated to physical address using this table. The page table translates virtual address to its physical counterpart by dividing the virtual address to find the corrosponding page table and page entry. Furthermore, the page directory address can be changed when switching between programs, meaning that programs will not be able to access the memory of other currently executing programs. An example of the page directory and table structure can be seen in figure 2.5.

Various flags can be set on a page directory and table entry, such as read-only, user mode page, etc. The flags make up the first 11 bits of an entry in the page directory and the page table. Because of this, each physical address must be on a 4kb alignment. The flags for an entry in the page directory and the page table can be seen in figure 2.6. These flag are used by the kernel to mark a page entry as present, allow usermode access and set the read/write permissions of a page. However, in order for the read only permissions to apply to the kernel also, then bit 16 must be set in the cr0 MSR.

User mode processes must be regulated when it comes to accessing kernel or other process memory as this could lead to sestive data being exposed or overided. Knowing this, it's easy to see how virtual memory can provide a way to regulate this kind of access. This is the reason why an architecture providing things like virtualisation is necessary to support an operating system. Due to the allowance of seperate, distinct memory between each process (and the kernel itself, which can be seen as its own process), a layer of security is implemented that hides the existance of processes from each other.

[7]This layout differs in 64 bit mode, as there is more depth to this structure.

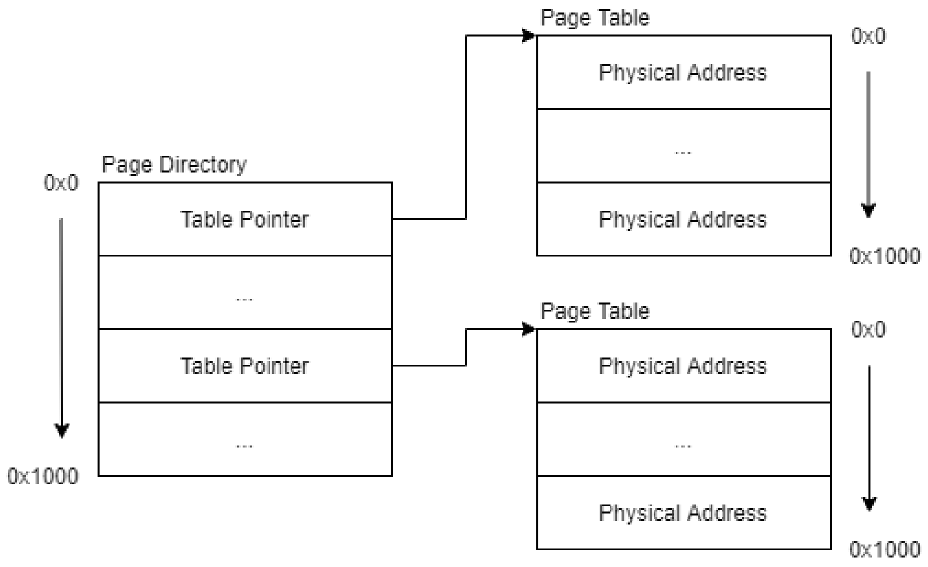
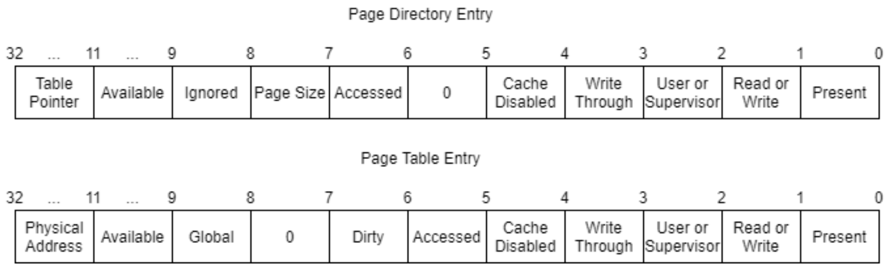


Figure 2.5: An example of the structure of page tables in a page directory.



Page Directory Entry

32 ... 11 ... 9 8 7 6 5 4 3 2 1 0

Table Pointer	Available	Ignored	Page Size	Accessed	0	Cache Disabled	Write Through	User or Supervisor	Read or Write	Present

Page Table Entry

32 ... 11 ... 9 8 7 6 5 4 3 2 1 0

Physical Address	Available	Global	0	Dirty	Accessed	Cache Disabled	Write Through	User or Supervisor	Read or Write	Present

Figure 2.6: The different bit flags for a page directory and table entry.

2.2.5 Video Graphics Array

The IA-32 architecture defines a Video Graphics Array (VGA) buffer, at the memory address of 0xB8000 (see 2.2.4 Virtual Memory, page 9), that can be used to output ASCII[8] text to the screen. Each entry in this buffer is 2 bytes wide, with the first byte being the ASCII letter to print and the second being the colour value of the letter. Moreover, the entries in the buffer directly corrospond to a place on the screen. By indexing the buffer like a grid, it becomes easy to set the location of a character. The general idea is to write strings to the buffer so they are displayed on screen. This functionality will allow for programs to display any textual output necessary, which is a must for an operating system.

[8]ASCII, or American Standard Code for Information Interchange, is a standard of encoding for characters in a computer system.

2.2.6 Global Descriptor Table

A memory structure known as the Global Descriptor Table (GDT) is used by the segment registers (see 2.2.2 Registers, page 8), to find the correct code and data selector. The code and data selector describe various features of the memory segment where the sgement registers point. For example, the access byte of the entry can describe if the segment is in user mode or kernel mode using a certain bit known as the *privilege* bit.

The act of using memory segmentation to seperate memory is now considered obsolete and has been replaced by paging. However, the GDT can still be used to define user mode and kernel mode. The GDT is stored in the 48bit GDT MSR and is set using the *lgdt* assembly instruction (figure 2.7).

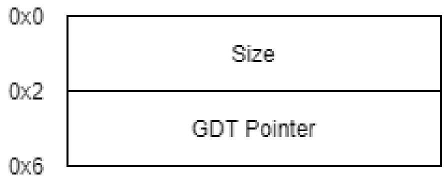


Figure 2.7: The structure of the GDT MSR.

The GDT MSR expects to receive a data structure that contains the address and size of the table. After performing this, the segment selectors can then be set to a byte offset into the GDT to define if the system is in user mode or kernel mode. For example, if the system wants to be in user mode the segment registers are set to a code a data selector with the user mode flag set.

Task State Segment

The GDT allows for an entry of a memory structure entry known as the Task State Segment (TSS), which is is described using the access byte of 0x89. The TSS allows for movement between kernel and user mode. Essentially this structure encodes the state of each register when a system call occurs (see 2.2.8 System Calling, page 14). The operating system can make use of this structure by setting the value of the SS segment register entry to the kernel data selector and the ESP (stack pointer) register entry of the structure to the address of the kernel stack. All other entries in this structure are optional. These values are then loaded into the corrosponding registers when a system call occurs. Because of this ability, a seperate kernel stack can be defined. This will reduce the likelihood of kernel data leaks and stack corruption when returning to user mode.

Once initialised with these two register values, the TSS can be added to the GDT. For changes to the TSS to take affect, the task register must be loaded with the GDT byte offset of the TSS, this is done with the *ltr* assembly instruction, while also setting the two bottom bits of the index, to signify user mode access.

2.2.7 Interrupts

Modern systems require that I/O devices, such as keyboards and sensors, receive servicing in an efficient manner so that little effect is taken on the throughput of the overall system. Before the introduction of interrupts, this was done via polling (or *"asking"*) the device to see if it would need to be serviced. It's simple to observe that this approach has detrimental effects on the overall performance of the system, as an approach of constantly looping and polling would have to be taken, thus decreasing the throughput of the system, this is known as "busy waiting". Taking this approach limits the number of tasks that the system can perform. This is where interrupts come in.

Interrupts is a more desirable method of handling I/O devices, as it allows the system to do whatever it wants until a device is ready to be serviced. Interrupts work by providing an asynchronus input that tells the processor to complete whatever instruction it is currently executing and then service the device that is requesting some. Once the servicing is complete, the processor can return to normal, providing its stack and registers have been restored.[9]

Furthermore, the processor also makes use of interrupts itself for things like exceptions, as this requires a handler to be called as soon as an exception is triggered, and to switch between RPLs of the CPU (see 2.2.3 Ring Priviliage Levels, page 8). Because of this, Interrupts are integral to usermode processes, as they allow for processes to request a service (e.g. printing to the screen, requesting keyboard input, etc) from the kernel. This is implemented via a structure in memory called the task state segment (TSS) that is part of the GDT (see 2.2.6 Global Descriptor Table, page 12) and is known as system calling (see 2.2.8 System Calling, page 14). The kernel can provide various handlers for each different system call to extend its functionality outside of ring 0. For this reason alone, interrupts are seen as an essential mechanism for the functionality of operating systems. And in the case of this project, will provide a large piece of functionaility to usermode processes.

Interrupt Descriptor Table

The Interrupt Descriptor Table (IDT) is a table of interrupt handlers for the IA-32 architecture. When an interrupt occurs, the table of handlers is offset with the interrupt number. The handler found at this offset is then given the necessary information it needs to handle the interrupt that was triggered. Like the GDT, the IDT has its own register and corrosponding instructions to manipulate this register. Once this register has been set with the address of the IDT, the system will then use this table to handle interrupts. The assembly instruction to set the IDT of the IA-32 architecture is *lidt*.

Interrupts must also be enabled via the *sti* assembly instruction after the table has been set. This instruction simply sets the interrupt flag in the EFLAGS register of the IA-32 architecture (see 2.2.2 Registers, page 8). If this flag is not

[9]More information about interrupts can be found in the Intel 8259A Programmable Interrupt Controller (8259A/8259A-2) datasheet.[6]

set, then interrupts are not enabled. Furthermore, when an interrupt occurs, care is to be taken to ensure that this flag is set after, as the interrupt flag is cleared upon entry to the interrupt handler.

Programmable Interrupt Controller

The 8259A Programmable Interrupt Controller (PIC) is a component within the IA-32 architecture that handles interrupts for hardware devices. Although being replaced by the advanced programmable interrupt controller (APIC) its commands can still be used to rebase the interrupt vector table (IVT) so that hardware interrupts do not interfere with exceptions[10]. Each core has its own local PIC, that can be accesed through port 0x20 and 0x21 using the *in* and *out* assembly instructions to read and write respectively. The global PIC can be found at port 0xA0 and 0xA1.

The IVT exists as an offset in the IDT and commonly the IVT must be rebased to avoid triggering the various exception interrupts defined in the IA-32 architecture; the PIC can be programmed to move the IVT offset.

2.2.8 System Calling

As discussed in previous sections, user mode programs like to be serviced by the kernel, as the kernel has elevated priviliages and can do more than a program running in user space. There are a few ways of doing this, however one of which is known as system calling.

A breif reminder that a system call is a call made by a program in user mode to communicate instructions and information with the kernel. When a system call occurs the central processing unit is immediately interrupted and told to handle the call, therefore they can be considered to be instantanious. If a particular interrupt was an system call, the processor switches to kernel mode and reads a structure in memory known as the task state segment (TSS), in order to set the values of the processor registers.

The IA-32 architecture provides many instructions to implement system calls and system call handling, a pair of which is the *int* and *iret* assembly instructions. The first instruction triggers an interrupt and the second returns from an interrupt handler.

This section may seem a bit redundant having just discussed interrupts, however, the IA-32 architecture provides a set of instructions that are faster than using the regular interrupt instructions to trigger a system call. The regular *int* and *iret* instructions must undergo multiple priviliage checks when switching between different RPLs and access memory in order to read the state of the TSS. This can add an unnecessary speed reduction to the system calls, something that should not be present in a fast operating system. Because of this, a new set of system call specific instructions was introduced by Intel through the Pentium II set of processors.

[10]This is because the APIC emulates the 8259A PIC

Defined in the IA-32 architecture is a set of fast system calls, *sysenter* and *sysexit*. The Intel developer manuals states that they "preform "fast" calls and returns because they force the processor into a predefined privilege level 0 state when SYSENTER is executed and into a predefined privilege level 3 state when SYSEXIT is executed. By forcing predefined and consistent processor states, the number of privilege checks ordinarily required to perform a far call to another privilege levels are greatly reduced. Also, by predefining the target context state in MSRs and general-purpose registers eliminates all memory accesses except when fetching the target code"[3]. These calls are labeled as faster than their *int* and *iret* counterparts as they avoid any interrupt overhead.

The IA-32 architecture provides three MSRs that must first be set before using the *sysenter* instruction:

- IA32_SYSENTER_CS
- IA32_SYSENTER_ESP
- IA32_SYSENTER_EIP

These registers are similarly used like the TSS, in which they set the state of the registers when a system call occurs, however, they are located within the processor itself, so can be retrieved in a more timely manner. The three registers set the *cs* segment register, the *esp* (stack pointer) register and the *eip* (instruction pointer) register, respectively. The purpose of these registers will be discussed in more detail later.

2.3 ELF File Format

The Executable and Linkable Format will be the operating system choice of program file. The ELF header contains various information about the file such as the magic number, a pointer to a program headers array and the entry point of the program. The magic number is used by many different file types to describe the type of file that it is, this value is usually found as the first four bytes of the file. For the ELF format, this value is 0x464C457F. The operating system can use this value to assists in checks that ensure that the image being loaded is in fact an ELF file.

Provided by the ELF header, is a pointer to a program headers array which is encoded as an offset into the file. The program headers array is an array of headers that describe the segments within the program file (figure 2.8). Using this array, certain information about these segments can be extracted such as the offset into the file of the segment, its size and its type. Again, these headers can be used by the operating system to parse and load a given ELF file.

The operating system itself will be built to an ELF format as it makes it easier for GRUB to boot the binary (see 2.4 GRUB Bootloader, page 16). As

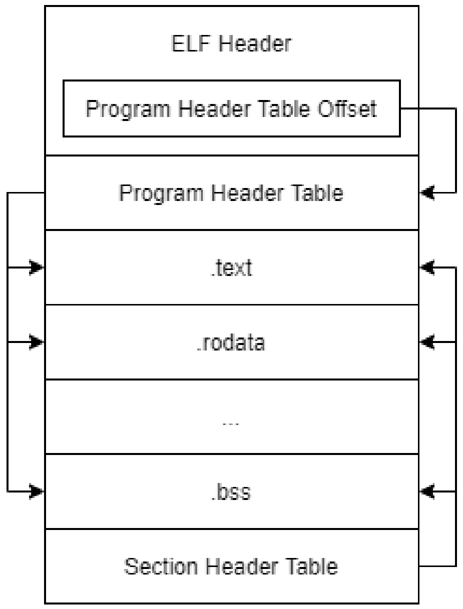


Figure 2.8: The structure of the segments within an ELF file.

the ELF file is simple to build and parse, it will make an optimal choice for the format used by programs within the operating system itself.[11]

2.4 GRUB Bootloader

The GNU GRUB manual describes a boot loader as "the first software program that runs when a compter starts. It is responsible for loading and transferring control to an operating system kernel software"[7]. In simplier terms, this is a bridge between the BIOS and the operating system. A bootloader will do the heavy lifting of gathering hardware information and getting the system ready for the kernel to execute. The GNU Project provides a well documented open source multiboot bootloader known as GNU GRUB that is simple to incorporate with any operating system.

GNU GRUB allows a user to load an operating system without recording the physical position of the operating system on the disk. All the user has to do is specify the name of the operating system, the drive and partition where the the operating system can be found - GRUB will then take care of the rest.

As this project declares no special cases, GNU GRUB can be used to boot this operating system. Utilising such a piece of software will greaty improve the speed in which this project can develop, as no work needs to be done to create

[11]More information on the ELF file format can be found in the Executable and Linking Format (ELF) Specification.[9]

a bootloader of its own.

2.5 C Programming Language

The C programming language is a ”system programming language” that has been adopted because of its simplicity when programming operating systems and compilers. It was designed originally by one man, Dennis Ritchie and is now one of the most widle used programming languages in the world. The high-level nature of the language and its lack of dependencies make it a good choice for this project.[12]

GNU provides a compiler library known as GCC that can be used to compile C and assembly programs to the IA-32 instruction set as an ELF file (see 2.3 ELF File Format, page 15). The library is known for its wide configuration settings and ease of use. The compiler can take many arguments, some of which can be used to tailor the output specifically for the IA-32 architecture. Due to this, the library that will be used to build this project is GCC.[13]

[12]More information on the C programming language can be found in The C Programming Language (Second Edition) by Brian W. Kernighan and Dennis M. Ritchie.

[13]More information on the GCC compiler library can be found at: https://gcc.gnu.org/.

Chapter 3

Design

3.1 Architecture

The architectural choice of the operating system is very important, as it can effect the whole development process and the outcome itself. An architecture is needed that will provide a base kernel for programs to execute at a high speed, without too many implementation choices. A monolithic kernel is said to be faster than a microkernel, so of course it is the optimal choice for this project (see 2.1.1 Monolithic, page 4). On top of this, if a look is taken at the microkernel, a general given is that the development time will take longer, as system features will have to be implemented in seperate binaries for user mode as well as providing base kernel features (see 2.1.2 Microkernel, page 5). One the other hand, a monolithic approach states that features handled by the kernel should be implemented in kernel mode. Making the monolithic approach the main choice for the architecture of this operating system will greatly decrease development time as everything must be implemented in a single binary, excluding user mode programs (this is also the case for the microkernel approach).

3.2 Memory Management

A main prerequisite of any modern sytem is the ability to manage the memory it provides. Memory management is an integral part to any system as it allows for allocation and deallocation of memory when a process so needs it. This prevents the need to declare static memory, which would entail predicting how much memory may be needed for execution whilst also being very costly - it can take up a lot of space in memory, whilst remaining unused for most of the time, thus starving other processes. Furthermore, a system that does not do utilise a memory manager can not take full advantage of paging and virtual memory. Know this, it is easy to see how a memory manager is a vital component to any modern day system. There are many methods of implementaing such a memory manager, a few are described in this section.

3.2.1 Bit Map

This apprach starts by dividing the usable memory into regions of n size. A table is then allocated that is used to track the allocation status of each page of memory, with each bit of the table representing a region of size n bytes. When a page is allocated, the corrosponding bit in the table is set to 1, meaning the memory is in use. The opposite is performed when a region is deallocated. This makes tracking of the various regions simple and memory efficient.

Because each page can easily be mapped to an offset in the bitmap, deallocation is constant time. However, as allocation consists of finding a page that is free by looking for a bit that is currently 0, allocation takes linear time[1]. A simple allocation and deallocation for a bit map of 6 regions is shown in figure 3.1.[2]

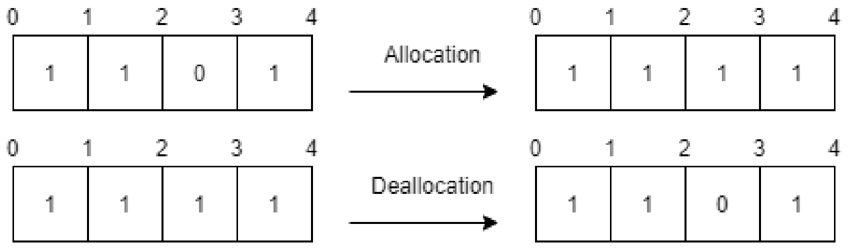


Figure 3.1: Memory allocation and deallocation using a bit map.

3.3 File System

The file system is a vital part to any operating system. It allows the operating system to store and retrieve files from a medium. As this operting system must be able to run programs, a file system will greatly help to store the programs and data that the operating system will need to execute. The file system must be fast to parse, as a requirement of this operating system is speed. A few different choices have been described below.

3.3.1 TAR Archive

The TAR file format offers a simplistic layout that is easy to parse. The file format consists of a sequence of blocks that are fixed to 512 bytes. Each block can represent a header, which describes a file in the archieve, or a block of data from a file within the archive. The header can be used to calculate the number

[1]Constant time and linear time are both ways to characterise the speed of an algorithm, with the latter taking longer than the former. Constant time is donoted as O(1) and linear as O(n), n being the highest number of iterations the algorithm can perform.

[2]More information on the bit map approach can be found at: http://www.cs.nuim.ie/ dkelly/CS240-05/Day 8 Slides.htm.

of blocks to jump to get to the next, using the size of the file that the header represents. This is because the blocks found after a header contain the data from the file the header points to (figure 3.2). Thanks to this, parsing a TAR archive is as simple as jumping block by block. This method being a simple iteration, the speed in which a file can be accessed is linear time. Assuming that the files stored in the TAR archieve are not compressed, it is easy to see how a TAR archive is an optimal choice for a file system design as it offers a comfortable and speedy parsing method.

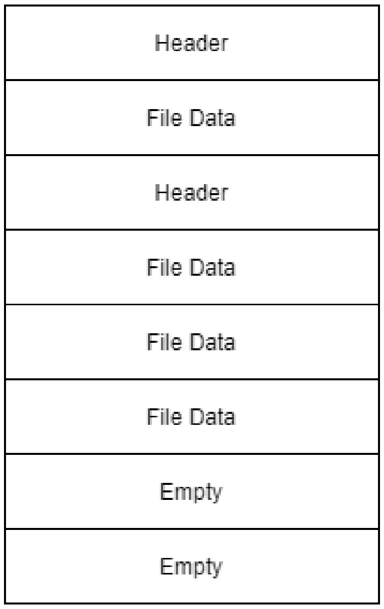


Figure 3.2: Example layout of blocks in a TAR archive that contains a root folder with one file.

Another benefit to incorporating a TAR file system is that file system can infact be preset with programs before the operated system is booted. Meaning that the operating system can come preinstalled with programs that can be set by the developer or a user looking to use the operating system.

As having a file system stored on the disk can slow down the speed of reading data, a small TAR file preset with all of the needed programs and data can be mapped into memory when the operating system is booted in order to ensure that reading from the file system is fast, as memory is faster than the disk. This supports the goal of speed within the operating system.

However, because of the layout of the TAR file, it will take great effort to manipulate the file system layout at run time, as blocks would need to be appended or moved. This would result in memory having to be preallocated to accommodate new files at run time. This will dramatically increase the size of the disk image which is not ideal when dealing with a lightweight portable operating system. Due to this, the TAR file only proves a benefit as a read only file system.[3]

[3]More information on the TAR file format specification can be found at: https://www.gnu.org/software/tar/manual/html_node/Standard.html

Chapter 4

Implementation

4.1 Project Setup

The structure of the project and the build system that it uses is important to achieving a functional operating system. These are described within this section, starting with the the directory layout of the project. Some prequisites before starting are the Linux operating system with an installation of GRUB and a GCC cross-compiler for the i686 processor.

4.1.1 Structure

The directory structure of the project is as follows:

- / contains the build script of the project.
- */source* includes all of the C source files, headers and linker files for the project.
- */source/asm* includes all of the assembly source files for the project.
- */source/disk* contains all files that will be added to the file system of the operating system.
- */source/drivers* contains all program sources and operating system specific headers.
- */source/grub* contains any configuration files needed for GRUB.

4.1.2 Compilation

This project, like any other project, needs to be compiled from its source into a disk image that can be booted by a computer system. This section describes the necessary tools needed to achieve this.

IA-32 Cross-Compiler

The C and assembly building library this project will use is GCC (2.5 C Programming Language, page 17). GCC, a compiler library made by GNU, by default, will build for the system it is executed on. In order to ensure that the project is built without any operating system dependencies, a cross-compiler must be built to handle the compilation. A cross-compiler will make certain that the project is built independent of the operating system it is compiled on.

For this project, GCC has been built to compile straight to the IA-32 architecture using the "i686-elf" target parameter. This will then be used throughout the project to carry out all compilation tasks.

As building a cross-compiler is a cumbersome task which depends on the library that is being used, it will not be discussed here. However, GCC provides its source code online. This can be built with various parameters to be tailored to a system. More information on this can be found on the GCC website.

For now it is assumed that a working cross-compiler is already set up. If this is not the case, GCC will build without operating system dependencies when supplied with certain parameters, please refer to the docs for more information on this.[1]

Now that a working cross-compiler is set up, certain files are needed to complete the compilation of the project, these are defined in the next sections.

linker.ld

The linker.ld file, that will be used by GCC to link all of the binary files of the kernel together, provides various information to the linker such as the layout of the kernel binary. The most important functionality of using such a script is the ability to declare certain memory locations that should be linked to a variable when the binary is linked and defining the layout of the kernel binary. Using such a functionaility allows for the kernel to know where certain sections of itself lie in memory. This ability can then be used later with the memory manager to reserve pages and map the kernel into virtual memory.

For example, the kernel entry function can be defined with:

```
ENTRY(__start)
```

Defining the default base address for the kernel can be done with:

```
. = 1M;
```

Aligning blocks of the kernel binary is as simple as:

```
.text BLOCK(4K) : ALIGN(4K)
```

And marking offset locations within the binary for use inside of the kernel:

```
kernel_start = .;
```

More information on how to structure a linker script can be found in the GCC documentation referred to in the previous section.

[1]https://gcc.gnu.org/onlinedocs/.

build.sh

A build script must be created to build the entire operating system and its dependencies into a disk image (or a *.iso* file). As the build of GCC that is being used exists on Linux, a bash script is declared to build all assembly files, C files and link them into one binary, the kernel.[2]

Commands can be used with the GCC cross-compiler to build to project. For example, building all source files using the linker script provided is as simple as:

```
i686-elf-as source/asm/*.s -o asm.o
i686-elf-gcc -c source/*.c -std=gnu99 -ffreestanding -O2 -Wall -
    Wextra
i686-elf-gcc -T source/linker.ld -o kernel.bin -ffreestanding -O2 -
    nostdlib *.o -lgcc
```

These commands will assemble all of the assembly files, compile all of the C file and then link them together using the linker script discussed in the previous section.

Once the kernel has been built, itself, grub and the *grub.cfg* file are added to the disk image, so that they can be booted by a system. The next section covers this.

4.1.3 Bootloading

Before the entry point of the operating system is called, a bootloader must be called to set everything up. In this case the bootloader is GRUB (see 2.4 GRUB Bootloader, page 16). This section briefly describes how GRUB can be implemented in a project.

grub.cfg

GRUB expects a configuration file that is added to the disk image so GRUB knows how to find the kernel binary. A basic layout of this configuration file can be seen below.

```
menuentry "kernel" {
  multiboot kernel.bin
}
```

The basic layout of the configuration file expect a menu entry, which can be anything, and a path to the kernel binary to boot, this will be a path relative to the *.cfg* file. The snippet above adds a menu entry to the GRUB screen that is displayed after boot, that will boot the *kernel.bin* binary.

build.sh

The configuration file created in the previous section can be added to a folder, for example *kernel*, along with the built kernel, and then made into a disk image using the command:

[2]More information on bash can be found at: https://www.gnu.org/software/bash/.

```
grub-mkrescue -o kernel.iso kernel
```

This command can be added to the build script and will produce an *.iso* file that can be used by a system to run the kernel. However, to use this command GRUB must be installed on the machine in which the kernel is being built on.

Alternatively, commands can be added to the bash script to pack everything together in a folder for this command to use. These need to be executed before the call to *grub-mkrescue* and after the project was compiled.

```
mv kernel.bin kernel/
mv grub.cfg kernel/
```

grub.h

GRUB defines a multiboot header that it will pass to the kernel entry point. A header file can be created and implemented that defines said multiboot header. The multiboot header contains a large amount of information about the machine, such as available memory regions, so it is of great use to the kernel itself. A limited definition of this multiboot header and memory mapping can be seen below.

```
typedef struct _MBINFO
{
  uint32_t flags;
  // ...
  uint32_t mmap_length;
  MAPINFO * mmap_addr;
} MBINFO;

typedef struct _MAPINFO
{
  uint32_t size;
  uint64_t base_addr;
  uint64_t length;
  uint32_t type;
} MAPINFO;
```

This multiboot header can be used later when creating a memory manager for the kernel (see 4.2.2 Memory Management, page 28).

boot.s

Now an assembly file can be created to handle booting into the kernel. GRUB will run this assembly file when it has set up the system. Before this, GRUB expects a multiboot header so it can tell it is running a valid binary. This is defined below and must be the start of the binary file, this can be done by adding the *.multiboot* block to the start of the linker script.

```
// Defines the multiboot header information.
.set MAGIC, 0x1BADB002
.set ALIGN, 1<<0
.set MEM,   1<<1
.set FLAGS, ALIGN | MEM
```

```
.set CHECKSUM,   -(MAGIC + FLAGS)

// Creates the multiboot header.
.section .multiboot
.align 4
.long MAGIC
.long FLAGS
.long CHECKSUM
```

Now a stack must be defined for the kernel entry point to use.

```
.section .bss
stack_bottom:
.align 16
.skip 16384
stack_top:
```

Once a stack is defined, a function can be implemented to call the entry point of the kernel. As the linker script has *__start* as the entry point, this is the name that will be used.

```
__start:
  // Adds the stack address to the stack pointer register.
  mov $stack_top, %esp
  // Adds the address of the Multiboot information data structure.
  push %ebx
  // Calls the kernel entry point.
  call kernel_main
  // ...
```

GRUB adds the address of the multiboot header to the *eax* register, so it has to be pushed on the newly created stack as an argument to the entry point of the kernel. Because of this, the entry point of the kernel must be defined as:

```
void kernel_main(MBINFO * mbinfo);
```

Now that the project structure and bootloader is set up, the kernel can start to be implemented.

4.2 Kernel

This section goes into detail around setting up specific features of a monolithic kernel. These following subsections will not cover the full implementation as that is simply too long. However, they do provide some insight on how to interact with such features of the IA-32 architcture in order to create a working kernel.

4.2.1 System Output

As mentioned previously, the IA-32 architecture defines a VGA buffer (see 2.2.5 Video Graphics Array, page 11). This subsection describes a basic implementation of *"stdio"*, the C standard input/output library, using this buffer to output strings to the screen. As this is a monolithic kernel, the library will be used within the kernel binary (see 2.1.1 Monolithic, page 4). Please keep in mind this is an example of how to interact with the VGA buffer and not the final product.

stdio.h

The first thing that need to be defined is the header file for the functions within the library to use. This header file will contain any memory structure definitions and macros to make interaction with the VGA buffer as simple as possible. The first definiton needed is the VGA buffer entry. The struct below defines this entry.

```c
typedef struct _vga_entry
{
  uint8_t character;
  uint8_t colour;
} __attribute__((packed)) vga_entry;
```

The character entry is the ASCII character that should be displayed on the screen and the colour entry is the colour of the character to display. The colour is not really important for the task at hand so the various colour definitions are skipped. Next comes the defintions of the buffer address and the maximum amount of columns and lines of the screen. Having these defined makes life easier when translating a screen position or checking for a newline.

```c
#define VGA_BUFF_ADDR 0xB8000
#define VGA_MAX_COLS  80
#define VGA_MAX_LINES  25
```

As mentioned previously, the VGA buffer can be indexed like a grid, however these grid coordinates must be traslated to an offset in the buffer to get the correct entry address. This can be done by multiplying the line count by the size of a screen column and then incrementing the result by the column number. A simple way to do this is by using a macro. Like so.

```c
#define VGA_GET_INDEX(L, C) ((L * VGA_MAX_COLS) + C)
```

To achieve textual output, the kernel will expose some functions that manipulate this buffer using these defintions to user mode programs as system calls. These functions *kprintf* and *putchar* can be used to append strings and characters to the screen buffer respectively. For now though they must be defined within the header file.

```c
void kprintf(const char * s, ...);
void putchar(char c);
```

stdio.c

A C file can now be implemented that contains the meat of the library. The first thing this file must do is set the address of the VGA buffer using the macro defined in the header file, using a global variable, so that is can be accessed by functions within the file. This opportunity can also be used to initialise the line and column indexes of the buffer also.

```c
static VGA_ENTRY * vga_buff = (VGA_ENTRY *)VGA_BUFF_ADDR;

static uint32_t line = 0, column = 0;
```

A simple implementation of *putchar* is defined below. Looking at these functions, it is easy to see how this design can act as a foundation to the *"stdio"* library.

```c
// Adds the character to the VGA buffer.
static inline void stdio_vga_put(uint8_t c, uint8_t col)
{
  vga_buff[VGA_GET_INDEX(line, column)].character = c;
  vga_buff[VGA_GET_INDEX(line, column)].colour = col;
}

// Goes to the next column or creates a new line.
static inline void stdio_vga_next(void)
{
  if (++column > VGA_MAX_COLS) { column = 0; line++; }
}

// Adds a character to the screen.
static inline void putchar(uint8_t c, uint8_t col)
{
  stdio_vga_put(c, col);
  stdio_vga_next();
}
```

A few details have been left out, such as checking for any special characters, which can be done using a switch case, and clearing the screen buffer, which should be done during initialisation. However, this is the basics of what is needed. The *kprintf* function can simply expand on this implementation by formatting and then looping through a string and calling *putchar* for each character.

4.2.2 Memory Management

Following the monolithic approach a memory manager must be implemented within kernel mode. The next few sections walk through some of the steps necessary to create such a memory manager. First starting with implementing a basic bit map memory allocation system and then moving to the creation of a paging system (see 2.2.4 Virtual Memory, page 9).

pfalloc.h

This kernel utilises the bit map memory management approach (see 3.2.1 Bit Map, page 19). The bit map, which will be shown later, is just an array of bytes. As it is not possible to define an array of bits in C, and as each bit represents a page, a mask will have to be used to set the correct bit of a byte in a bit map.

A start to this is defining a header file that stores all of the useful declarations for the manager. For example, the manager needs to be able to translate a page number into a mask and translate a page number to an offset in the bit map. This can be done with two simple macros.

```
#define SET_PAGE_MASK(  N ) (0x1 << ((N % 8)))
#define PAGE_TO_BYTE( N ) (N / 8)
```

Because there are eight bits in a byte, the page mask is generated by using the modulo eight of a page number to get the remainder of dividing the page number by eight, this remainder is then used to shift a bit so that the positivite bit is at the correct offset of the byte in the bit map. The second macro divides a page number by 8 to get the index into the array of bytes which is the bitmap. Applying these macros is shown in the next section.

The size of a memory page can also be defined. This will come in useful later when the need to translate a page number to a physical memory address is required.

```
#define PAGE_SIZE  4096
```

pfalloc.c

The manager will calculate the number of possible pages in memory at run time and then allocate a single page of these to be the bitmap for the memory manager to use. It does this by using the mutiboot header provided by GRUB to find the size of the memory and dividing the size by the *page_size* definition to get the number of possible pages in memory (see 4.1.3 Bootloading, page 24). This length is then divided by eight and stored by the memory manager so it can be used later for iteration of the bytes in the bit map.

As the main focus of this section is a bit map memory manager and implementing this is not strictly necessary, details on how to use the multiboot header are omitted. Implementation of this can be derived from the GRUB documentation.[3]

Following this, the manager ensures to reserve any protected memory regions, such as the page that has been used for the bit map, the kernel and any that have been marked by GRUB as reserved via the multiboot header.

Setting a page as reserved is as simple as setting the bit of a byte, therefore the mask this is generated generated using the page number is ORed with the original byte stored in the bit map. The function below takes the page number to reserve and set the bit of the bit map to high, to indicate that the page is in use.

```
void pfalloc_set(uint32_t page_num)
{
  // Sets the page as used.
  pages[PAGE_TO_BYTE(page_num)] |= SET_PAGE_MASK(page_num);
}
```

This function can now be used with a page number to reserve a certain page in the bit map. Later it will be explained how an address can be used for this function as well as a page number.

[3]https://www.gnu.org/software/grub/manual/grub/grub.html.

Releasing a page is the same concept apart from the page mask is XORed with 0xFF to filp the mask (e.g. if a mask was 0b10000, it would become 0b11101111) and then ANDed with the bit map byte entry to ensure that all bits are unchanged apart from the page that is being released.

```
void pfalloc_release(uint32_t page_num)
{
  // Sets the page as free.
  pages[PAGE_TO_BYTE(page_num)] &= (0xFF ^ SET_PAGE_MASK(page_num))
    ;
}
```

Now that it is possible to set and release pages of the bit map, other functions like searching the bit map for the first free page can be implemented. This function will allow the memory manager to find the first free page to allocate when requested.

```
uint32_t pfalloc_first_avail(void)
{
  // Iterates through the bit map bytes.
  for (int i = 0; i < length; i++)
  {
    // Iterates through the bits in each byte.
    for (int b = 0; b < 8; b++)
    {
      // Is this a free page?
      if ((~pages[i] >> b) & 1)
      {
        // Returns the page number.
        return (i*8)+b;
      }
    }
  }
}
```

The manager should then provide a publid function, *pfalloc_alloc*, to find an allocate a free page in memory. The start of an implemetation of the former is shown below.

```
void * pfalloc_alloc(void)
{
  // Find an available page.
  uint32_t page_num = pfalloc_first_avail();
  // Reserve the page.
  pfalloc_set(page_num);
  // ...
}
```

As the size of a page is known by the manager, it can be used to translate a page number to it corrosponding physical address. This is done simply by multiplying the page number by the page size. Division can be used to reverse the translation in order to translate a page base address back to a page number.

```
uintptr_t page_addr =   page_num * PAGE_SIZE;
uintptr_t page_num =    page_addr / PAGE_SIZE;
```

As the monolithic approach is considered, such a function, along with a page-to-address translation, can be used throughout a kernel whenever a memory page is needed. This will save on the size of the kernel binary and allow for dynamic memory to be requested. Knowing this, it easy to see how these building blocks can be used to achieve a functioning memory manager for this kernel.

paging.h

Starting with the paging system for the kernel, a header file must be defined to declare some macros for the system to use. Below is an example of the definitions that the paging system will use to manage pages in the page directory and table. The size of a page in bytes is defined, the max number of entries in a directory and table is defined, along with the amount of bytes that a page table manages and a mask used for page directories to remove any flags from an entry.

```
#define PAGE_SIZE              4096
#define MAX_PAGETABLE_ENTRIES  1024
#define SIZE_OF_MEM_REGION     (MAX_PAGETABLE_ENTRIES * PAGE_SIZE)
#define GET_PAGE_TABLE_MASK    0xFFFFFF00
```

As there are flags at the lower bits of a page directory, a macro can be defined to translate an address in the page directory to a valid page table address. By passing it an entry from the page directory, it will use the mask to remove any flag bits, making it into a valid pointer. This will come in handy later.

```
#define GET_PAGE_TABLE(ENTRY)  ((uint32_t *)(ENTRY &
    GET_PAGE_TABLE_MASK))
```

Specific flags for a page directory and table entries can be declared so certain priviliages can be applied to pages. These can be applied to an entry using the bitwise *"OR"* operator.

```
#define PAGE_PRESENT   0x1
#define PAGE_RW        0x2
#define PAGE_USER      0x4
```

paging.c

The meat of the paging system is defined in this section. The page directory can be created using the *pfalloc_alloc* function defined earlier in the memory manager, as a page is the correct size for the table. Because of this, multiple page directories are able to be allocated, meaning that programs can have seperate virtual memory.

```
uint32_t * page_directory = (uint32_t *)pfalloc_alloc();
```

To ensure that there is no data that can cause conflicts present in this page table, the region must be set to zero before continuing. This page directory should be initialised with a table when a table is required for a virtual address, this can also be done using the same function, *pfalloc_alloc*. For example, allocating the fifth page table is as simple as.

```c
page_directory[4] = (uint32_t *)((uint32_t)pfalloc_alloc() |
    PAGE_PRESENT);
```

A page is allocated and the address has its present flag set. Other access modifiers defined in the header can be applied on top of this allocation using the bitwise OR. Once the page directory is set up, a function can be defined to map a physical address to a virtual one. The code snippet below also demonstrates how to translate a virtual address to its page directory offset, which is useful for allocating a table with a given virtual address. A simple way of doing this is checking for the present bit in the page directory entry, and then allocating the table if this bit is not set, however this is not shown here.

```c
void paging_map_page(uint32_t virtual, uint32_t physical, uint32_t
    flags)
{
  // Checks if the addresses are 4KB aligned.
  if (((virtual % PAGE_SIZE) != 0) && ((physical % PAGE_SIZE) != 0)
    ) return;
  // Calculate the offset into the directory for the virtual
    address.
  uint32_t dir_off = virtual / SIZE_OF_MEM_REGION;
  // Calculates the offset into the page table.
  uint32_t tab_off = (virtual % SIZE_OF_MEM_REGION) / PAGE_SIZE;
  // Allocate a table if necessary.
  // ...
  // Maps the physical page to the virtual one.
  GET_PAGE_TABLE(page_directory[dir_off])[tab_off] = physical |
    flags;
}
```

First a check must be made to ensure that the addresses given are aligned to 4 kilobytes as that is the size of a page in the system. Then the virtual address is divided by the memory size of a page table to find the correct entry in the page directory.[4] After finding the page directory offset, the page table offset can be calculated by using the remainder of this calculation and the result of dividing it by the size of a memory page, as this will map to an index in the page table. Using the macro defined earlier, the page directory can be indexed to get the table pointer, which is then indexed with the table offset to get the correct entry. After this, the physical address is be ORed with the given flags defined in the header to set the correct priviliages of the page and added to the entry in the page table.

If translation is needed to translate a virtual to physical address, it can be done via the code snippet below. It functions much like the previous function, however returns the page table entry instead of setting it.

```c
uint32_t paging_virtual_to_physical(uint32_t virtual)
{
  // Calculate the offset into the directory for the virtual
    address.
```

[4]The memory size of a page table is how much memory a page table can map in bytes overall, which is why it can be used to divide a virtual address to its corrosponding page table entry.

```
  uint32_t dir_off = virtual / SIZE_OF_MEM_REGION;
  // Calculates the offset into the page table.
  uint32_t tab_off = (virtual % SIZE_OF_MEM_REGION) / PAGE_SIZE;
  // Calculates the remainder to add to the physical address
  uint32_t rem = (virtual % SIZE_OF_MEM_REGION) % PAGE_SIZE;
  // Check if the table is present.
  // ...
  // Check if the physical address is present.
  // ...
  // Returns the physical address.
  return (GET_PAGE_TABLE(page_directory[dir_off])[tab_off] &
    GET_PAGE_TABLE_MASK) + rem;
}
```

This function becomes useful when the need to upmap a virtual address or map a virtual address to a different virtual address arises. To set the page directory address and enable paging, two assembly functions can be defined.

```
__enable_paging:
  mov %cr0, %eax
  or $0x80000000, %eax
  mov %eax, %cr0
  ret

__set_page_dir:
  mov 4(%esp), %eax
  mov %eax, %cr3
  ret
```

It must be noted that before enabling paging, the kernel must be mapped into virtual memory, or else the system will crash. This can be done using a one-to-one mapping for now, so that the kernel does not have to be rebased. Paging must also only be enabled after setting the page directory address. Using this information and code snippets can be enough to set up basic paging for the IA-32 architecture.

4.2.3 Interrupts

Before interrupts can be implemented, the GDT has to be created and initalised (see 2.2.6 Global Descriptor Table, page 12). This is because each entry in the IDT expects a reference to the code segment where the interrupt handler lies, this can be in user mode or kernel mode (see 2.2.7 Interrupts, page 13). A code and data segment are simply references to the memory segment where the code and data will lie. The GDT can also be used along side the TSS to switch to user mode and allow for system calls. The next two sections detail the basic implementation of the GDT with a TSS, followed by two sections on the implementation of a basic IDT and concludes with a breif two sections on the PIC.

gdt.h

The GDT header file should define the various memory structures that are needed by the GDT. A GDT is a table of entries, so its only right that the first thing defined is the structure of an entry in the GDT.

```
typedef struct _GDT_ENTRY
{
  uint16_t limit_low;
  uint16_t base_low;
  uint8_t base_mid;
  uint8_t access;
  uint8_t flags_limit_high;
  uint8_t base_high;
} __attribute__((packed)) GDT_ENTRY;
```

These structure allows for the defintion of a base address for the segment, the size of the segment and any access modifiers of the segment. As segmentation is no longer used to segment memory, the flags and the access attributes are the most important fields. Certain flags can be defined for an entry in the GDT, they are shown below.

```
#define GDT_PAGE_GRAN 8
#define GDT_PROC_32 4
```

One purpose of the access byte is to provide a flag to determine if the segment is in user mode or kernel mode. The code and data selectors access bytes for user mode and kernel mode along with the TSS are defined below. These will be used when adding an entry to the GDT.

```
#define SEL_KER_CODE 0x9A
#define SEL_KER_DATA 0x92
#define SEL_USR_CODE 0xFA
#define SEL_USR_DATA 0xF2
#define SEL_TSS 0x89
```

As some attributes hold more than one piece of data, the liberty can be taken to define a few extra macros to make inserting into the GDT entry a bit easier. These will shift the given values by a certain index size so they can be added to the correct part of the attribute. Their use be realised after a look is taken at the *gdt_add_entry* function later.

```
#define SHORT_INDEX(I, S) ((I >> (S * 16)) & 0xFFFF)
#define BYTE_INDEX(I, S)  ((I >> (S * 8)) & 0xFF)
#define FLAGS_LIMIT_HIGH(LIMIT, FLAGS) ((BYTE_INDEX(LIMIT, 2) & 0xF
    ) | ((FLAGS & 0xF) << 4))
```

Furthermore, a defintion of the GDT information structure is needed, this has to be set to the size and base address of the GDT created and passed into the GDT MSR described in the next section.

```
typedef struct _GDT_INFO
{
  uint16_t limit;
  uint32_t base_addr;
} __attribute__((packed)) GDT_INFO;
```

Finally, the TSS entry must be defined. As this will be used to set the state of the registers for software interrupts in the future.

```c
typedef struct _TSS_ENTRY
{
  uint32_t prev_tss;
  uint32_t esp0;
  uint32_t ss0;
  uint32_t esp1;
  uint32_t ss1;
  uint32_t esp2;
  uint32_t ss2;
  uint32_t cr3;
  uint32_t eip;
  uint32_t eflags;
  uint32_t eax;
  uint32_t ecx;
  uint32_t edx;
  uint32_t ebx;
  uint32_t esp;
  uint32_t ebp;
  uint32_t esi;
  uint32_t edi;
  uint32_t es;
  uint32_t cs;
  uint32_t ss;
  uint32_t ds;
  uint32_t fs;
  uint32_t gs;
  uint32_t ldt;
  uint16_t trap;
  uint16_t iomap_base;
} __attribute__((packed)) TSS_ENTRY;
```

gdt.c

Now that the defintions are in place, code can be written to add entries to the GDT. The function *gdt_add_entry*, defined below, can be used to do this.

```c
void GDT_add_entry(GDT_ENTRY * entry, uint32_t base, uint32_t limit
    , uint8_t access, uint8_t flags)
{
  // Adds the limit.
  entry->limit_low =        SHORT_INDEX(limit, 0);
  // Adds the flags.
  entry->flags_limit_high = FLAGS_LIMIT_HIGH(limit, flags);
  // Adds the base.
  entry->base_low =         SHORT_INDEX(base, 0);
  entry->base_mid =         BYTE_INDEX(base, 2);
  entry->base_high =        BYTE_INDEX(base, 3);
  // Adds the access byte.
  entry->access =           access;
}
```

The function initialises an entry in the GDT with the specified parameters.

Now that adding entries to the GDT is a simple process. The below snippet shows the basic layout of the GDT. The IA-32 expects the first selector to be blank, so it must be initialised to null. The next two entries are the code and data selector for the kernel. Following this, is the user mode code and data selector. These do not need to be in any particular order, however their offsets must be noted for later use.

```
GDT_add_entry(&table[0], NULL, NULL, NULL, NULL);
GDT_add_entry(&table[1], 0x0, 0xFFFFFFFF, SEL_KER_CODE,
    GDT_PAGE_GRAN | GDT_PROC_32);
GDT_add_entry(&table[2], 0x0, 0xFFFFFFFF, SEL_KER_DATA,
    GDT_PAGE_GRAN | GDT_PROC_32);
GDT_add_entry(&table[3], 0x0, 0xFFFFFFFF, SEL_USR_CODE,
    GDT_PAGE_GRAN | GDT_PROC_32);
GDT_add_entry(&table[4], 0x0, 0xFFFFFFFF, SEL_USR_DATA,
    GDT_PAGE_GRAN | GDT_PROC_32);
```

As you can see, the first parameter is a pointer to an entry in the GDT table, which is just a global array of six GDT entries, the base and limit paramaters can just be set to 0x0 and 0xFFFFFFFF respectively as they are not important, the other attributes define the priviliages of each segment.

Before going any further, it is important to set up the TSS for user mode interrupt purposes. First a global TSS structure must be defined and initialised to zero, in this case it is called *tss*. The *ss* register field is to be set to the byte offset of the kernel data segment selector and the *esp* register field is to be set to the address of the kernel stack, which should be defined globally.[5] This will be now be the state of the registers if a software interrupt should occur.

```
tss.ss0 = 0x10;
tss.esp0 = &stack_top;
GDT_add_entry(&table[5], &tss, sizeof(tss), SEL_TSS, NULL);
```

Once initialised with these two register values, the TSS is added to the GDT. For changes to the TSS to take affect, the task register must be loaded with the GDT offset of the TSS, this is done with the *ltr* assembly instruction, while also setting the two bottom bits of the index (specifying an RPL of 3), to signify user mode access.

```
__tss_flush:
    mov $0x2B, %ax
    ltr %ax
    ret
```

After the table is fully initialised, the table addess and size must be added to the GDT information structure in order to be set on the MSR.

```
// Creates the GDT information struct.
GDT_INFO info;
// Sets the size of the GDT.
info.limit = sizeof(table) - 1;
// Sets the linear address of the GDT.
info.base_addr = (uint32_t)&table;
```

This strcture must be added to the GDT MSR using the *lgdt* instruction. This can be done with a small assembly function that takes one argument, the address of the GDT information structure and moves it into the GDT MSR.

[5]The segment selector in this case is 0x10, and can be calculated using the size of the *gdt_entry* structure.

```
__set_GDT:
  mov 4(%esp), %eax
  lgdt (%eax)
  ret
```

In order for changes to the GDT to take effect, the segment registers must be reloaded, this is done with a simple piece of assembly code that sets all of the segement registers to the kernel mode segment selectors using the byte offset discussed before.

```
__reload_seg_regs:
  ljmp $0x08, $reload_cs
reload_cs:
  mov $0x10, %ax
  mov %ax, %ds
  mov %ax, %es
  mov %ax, %fs
  mov %ax, %gs
  mov %ax, %ss
  ret
```

These functions along with the TSS flush defined earlier can now be used to finialise the GDT set up. Calling these functions is as simple as calling any regular C function and should be called in the order defined below to avoid crashing the system.

```
// Uses LGDT to set the GDT register.
__set_GDT(&info);
// Reloads the segment registers.
__reload_seg_regs();
// Flushes the TSS.
__tss_flush();
```

idt.h

The IDT set up behaves much the same as the GDT, so not too much detail is necessary. An entry in the IDT is defined as follows.

```
struct IDT_ENTRY
{
  uint16_t offset_low;
  uint16_t selector;
  uint8_t zero;
  uint8_t type_attr;
  uint16_t offset_high;
}
```

This entry allows for the definition of a handler address, the segment selector that this handler uses, which is a byte offset into the GDT, and the attributes of the handler, such as which type of interrupt it handles; known as the gate type. For example, an interrupt or trap, which can be defined with the following.

```
#define GATE_INT   0xE
#define GATE_TRAP 0xF
```

Certain other attributes, such as if the interrupt handler be called from a user mode interrupt and if the interrupt handler entry is present can be defined also. These flags will be applied to an entry in the IDT to define certain attributes of the entry.

```c
#define ATTR_PRESENT    ((1 & 1) << 7)
#define ATTR_USER       ((3 & 3) << 5)
#define ATTR_STORAGE    ((0 & 1) << 4)
```

Last but not least, the IDT information structure to be passed into the IDT MSR must be defined. This follows the same layout of the GDT information structure.

```c
typedef struct _IDT_INFO
{
  uint16_t limit;
  uint32_t base_addr;
} __attribute__((packed)) IDT_INFO;
```

idt.c

A simple implementation of a function to install a handler can be viewed below.

```c
void IDT_add_entry(IDT_ENTRY * entry, uint32_t offset, uint16_t
    selector, uint8_t gate_type)
{
  // Adds the address of the routine.
  entry->offset_low = SHORT_INDEX(offset, 0);
  entry->offset_high = SHORT_INDEX(offset, 1);
  // Adds the selector.
  entry->selector = selector;
  // Zeros the unused part.
  entry->zero = NULL;
  // Sets the attributes of the interrupt.
  entry->type_attr = (gate_type & 0xF) | ATTR_USER | ATTR_STORAGE |
     ATTR_PRESENT;
}
```

This function takes an address of an entry in the IDT and initialises it with the given parameters. The first parameter is a pointer to the handler function for the interrupt, the second is the segment selector to use and the third is the gate type. The function also by default makes the handler available to user mode interrupts. This is for simplicity purposes and can be changed.

Declaring an interrupt handler must be done in assembly as the *iret* instruction needs to be used to clean up any overhead. Care is also taken to disable and reenable interrupts after the handler has been called. This can be done with a simple wrapper function macro for the GCC assembler.

```
// Macro to define interrupt routines.
.macro irtn f
.global __\f
.type __\f, @function
__\f:
  // Clear interrupt flag.
  cli
  // Call C handler.
  call \f
  // Set interrupt flag.
  sti
  // Return from the interrupt.
  iret
.endm
```

This macro takes an interrupt handler function name defined in C and then produces an assembly function with a "__" prefix that wraps the original function. For example, a function named *double_fault* would have a wrapper called *__double_fault*.

Certain other abilities should be added to such a macro, such as register preservation, so that no data is lost or leaked from the kernel. However, this is just a simple example.

An example of installing an interrupt handler using the IDT can be seen below, with table being a global pointer to the IDT. As first entries of the table are exception handlers, a double fault handler can easily be installed to catch any system crashes.

```
IDT_add_entry(&table[0x8], ((uint32_t)&__double_fault), 0x8,
    GATE_INT);
```

As the double fault handler is located at the 0x8 entry, defined in the Intel developer manuals, this is where the handler entry will be installed. The kernel code selector defined in the GDT must be used for an interrupt handler in kernel space, in this case the byte offset is 0x8.

Initialising the IDT information structure is the same as the GDT, so there is no need to show it here. The only difference is when setting the IDT MSR, the *lidt* instruction is to be used instead.

Interrupts must also be enabled via the *sti* opcode after the table has been set. This instruction simply sets the interrupt flag in the EFLAGS register of the IA-32 architecture (see 2.2.2 Registers, page 8). If this flag is not set, then interrupts are not enabled. This can be implemented with another small assembly function.

```
__sti:
  sti
  ret
```

pic.h

The PIC is used to handle any hardware interrupts. The header file must define the port address of the local and global PIC, so commands and data can be sent to each. This is done with the below defintions.

```
#define PIC1          0x20
#define PIC2          0xA0
#define PIC1_COMMAND  PIC1
#define PIC2_COMMAND  PIC2
#define PIC1_DATA     (PIC1+1)
#define PIC2_DATA     (PIC2+1)
```

Various configuration parameters must also be defined. These will be used when setting up and rebasing the local and global PIC.

```
#define ICW1_ICW4 0x01
#define ICW1_INIT 0x10
#define ICW4_8086 0x01
```

pic.c

The local and global PIC can be communicated with using ports. Ports are essentially just addresses for the IA-32 I/O bus and have to be read and written to using specific assembly instructions. The assembly code to read and write ports can be seen below.

```
__read_port:
  mov 4(%esp), %dx
  in %dx, %al
  ret

__write_port:
  mov 4(%esp), %dx
  mov 8(%esp), %al
  out %al, %dx
  ret
```

These functions will read and write to ports respectively.

A mask can be applied to the local and global PIC to only allow certain hardware interrupts, such as the keyboard. This is done using the Interrupt Mask Register. Appying a mask to the Interrupt Mask Register is as simple as.

```
__write_port(PIC1_DATA, 0xFD);
```

This mask disables all hardware interrupts for the first PIC apart from the keyboard interrupt, which will be implemented in the next section as a demonstration of a hardware interrupt (see 4.2.4 Keyboard Input, page 41).

As the IVT is typically at the start of the IDT, it will trigger the various exception interrupts defined in the IA-32 architecture, so the PIC is programmed to move the IVT to the IDT offset of 0x20. The process of rebasing the local and global PIC is as follows:

```
void pic_init(void)
{
  uint8_t pic1_mask = __read_port(PIC1_DATA);
  uint8_t pic2_mask = __read_port(PIC2_DATA);

  __write_port(PIC1_COMMAND, ICW1_INIT | ICW1_ICW4);
  __write_port(PIC2_COMMAND, ICW1_INIT | ICW1_ICW4);

  __write_port(PIC1_DATA, 0x20);
  __write_port(PIC2_DATA, 0x28);

  __write_port(PIC1_DATA, 4);
  __write_port(PIC2_DATA, 2);

  __write_port(PIC1_DATA, ICW4_8086);
  __write_port(PIC2_DATA, ICW4_8086);

  __write_port(PIC1_DATA, pic1_mask);
  __write_port(PIC2_DATA, pic2_mask);
}
```

The 8259A PIC also requires that interrupt handler lets the controller know when they are done, by writing 0x20 to the local or global PIC, depending on which I/O interrupt was triggered. This can be done with:

```
__write_port(PIC1, 0x20);
```

This function should be used along with the keyboard handler in the next section to successfully return from an interrupt.

4.2.4 Keyboard Input

A key part to an operating system is user input. In order for this to be functional operating system, a driver much be implemented to handle keyboard input. This is a simple task as the IDT and PIC are already configured. The PS/2 driver implements its functionailty by first enabling itself on the PIC using an interrupt mask, as shown the the previous section (see 4.2.3 Interrupts, page 33).

Depending on the keyboard, it can be using one of three different scan codes sets. For the sake of functionality, this operating system only implements one scan code set, the second one. The scan codes values in these sets are more or less arbitrary, so a secondary mapping must occur to map the scan code to a more maskable counterpart. Providing this will make it easier to extract certain information on the scan codes, such as if it was a key press or release, if shift was being held at the time of the key press, or even the keys position on the keyboard. These secondary scan codes are two bytes in length, that way the key value can be stored along with various flags that determine the state of the key. Figure 4.1 details the layout of the bits and flags in the translated secondary scan codes.

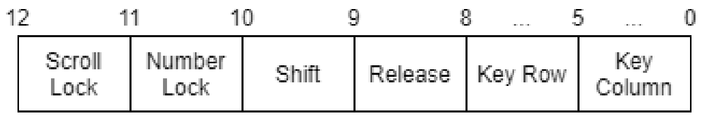


Figure 4.1: The layout of bits within the secondary scan code set.

keyboard.h

Reading data from the PS/2 keyboard is done using the read and write port functions defined in the previous section. Because of this, the port number of the keyboard must be defined.

```
#define PS2_PORT_DATA 0x60
```

For now the other flags of the new scan code set will remain ununsed, apart from the shift and present flag. Below the *shift_down* definition will be used to set the shift flag of the translated scan code.

```
#define SHIFT_DOWN 0x2
```

keyboard.c

To read a keypress from the keyboard buffer the function below can be called.

```
uint8_t scan_code = __read_port(PS2_PORT_DATA);
```

After a scan code has been read from the keyboard buffer the below code segment can be used to translate the scan code into a secondary scan code set. This is done using a mapping table, the definition of this table is not important as implementation depends on what is trying to be achieved, in this case it is adding simplicity to parsing scan codes. The mapping table translates each scan code to a scan code that can be masked to get its row and column on the keyboard.

```
void keyboard_handler(void)
{
  // ...
  // Is true if we are looking at a key release.
  bool release = ((scan_code - 0x80) > 0x0) ? 1 : 0;
  // Converts the scan code into something more understandable.
  uint32_t code = mapping[release ? (scan_code - 0x80) : scan_code
    ];
  // Has a shift key been pressed?
  switch (code)
  {
    case 0x80:
      // Shift pressed.
      shift_mask = (release ? (shift_mask ^= SHIFT_DOWN) : (
    shift_mask |= SHIFT_DOWN));
      break;
    default:
      // Adds the shift maks to the scan code.
      code |= ((shift_mask | release) << 8);
      break;
  }
  // ...
}
```

When a keyboard interrupt occurs, the first thing the driver does is read the scan code of which key was set. It then finds out if the key was a key press or a release. Once it has done this, it can then use the scan code within a mapping table to translate the scan code into a secondary scan code and then set the shift and key release flags. After it has completed this, the keyboard input is then buffered in wait for a program to request some input.

The handler function has to be wrapped with the assembly wrapper defined in the previous section, producing *__keyboard_handler*. After doing so, the handler now needs to be installed in the IDT. As the IVT table was rebased in the previous section, the keyboard interrupt lies at the 0x21 interrupt handler. Once it has added the handler to the 0x21 entry of the IDT, it can then wait for a keyboard interrupt to occur.

```
IDT_add_entry(&table[0x21], (uint32_t)&__keyboard_handler,
    DEF_CODE_SEL, GATE_INT);
```

Now that it is possible to receive and remap scan codes, they can be buffered and passed to user programs in the future via system calls (see 4.2.6 System Calls, page 45).

4.2.5 Programs

The main aim of this operating system is to provide the ability to run programs while also gauaranteeing a safe environment for doing so. This means that the programs must execute in user mode, using system calls to perform elevated tasks, such as reading keyboard input. For programs to be able to call system calls they must have a header library that defines each system call.

Before implementing system calls (see 4.2.6 System Calls, page 45), the basic layout of a program must be defined. The basic structure of how a program can use a kernel library to include system functions is shown in figure 4.2.

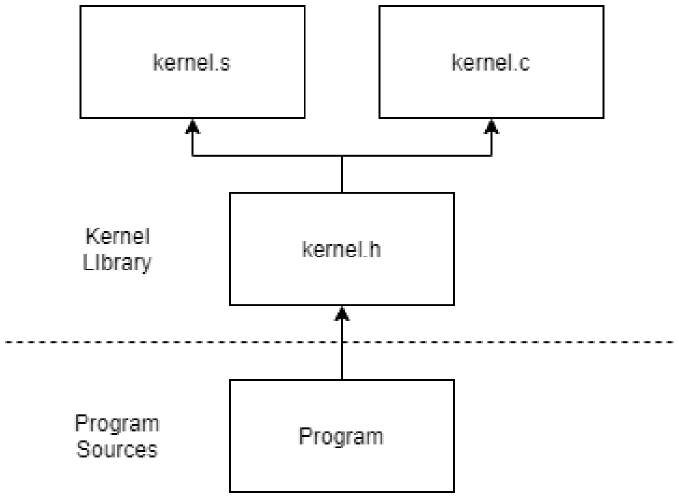


Figure 4.2: The basic structure of an operating system program.

The basic layout consists of a assembly file to implement the direct system calls and a header file which contains system call declarations for a program to include. The C source file can be used to include C wrappers for certain system calls if necessary, for example, formatting a string before sending it to *printf*.

This is a basic structure that will server for the purposes of this project, however, a more complex library should be built when dealing with a more complex operating system.

When compiling such programs for the operating system, attention should be paid to details such as the base address defined by the linker. The reason for this is because it must be known by the image loader (see 4.2.8 Image Loading, page 50).

kernel.s

This assembly file is a bit different, it is not built with the kernel, instead it is built with the programs to run on the kernel. Its use should be being included with a program that is being compiled for this operating system. The macro below will define an assembly function to call a specific system call.

```
// Macro to create a new system call.
.macro sysdef name id=0
  .global \name
  .type \name, @function
  \name:
    // Arguments.
    mov 4(%esp), %ebx
    mov 8(%esp), %esi
    mov 12(%esp), %edi
    // The system call id to invoke.
    mov $\id, %eax
    // Tells the kernel the return address.
    pop %edx
    // Passes the user stack to the kernel.
    mov %esp, %ecx
    // Trap to the kernel.
    sysenter
.endm
```

This macro will create a function with the given name that calls the specified system call ID. Without going into too much detail, it provides three arguments to a system call via the *ebx*, *esi* and *edi* registers, adds the system call ID to the *eax* register and then preserves the return address and user stack. The registers used here are arbitrary, and have to be handled by the system call handler defined in the next section. The stack cannot be used for arguments, the reason for this is decribed there also.

Now a function can be created to call *kprintf*. This is shown below.

```
sysdef printf 0x0
```

Any ELF binary that is built with this assembly file will be able to access the *printf* system call, which calls the handler function with an ID of 0x0, meaning the first entry in the table.

After a program is built it must be added to the file system to be loaded by the kernel. This is described in the coming sections (see 4.2.7 File System, page 47).

Now that the basic implementation of defining an ID and calling *sysenter* has been defined, system calls can be implemented.

4.2.6 System Calls

The operating system includes various header files for programs to be compiled with (see 4.2.5 Programs, page 43). These header files provide system calls for programs to communicate with the kernel. As mentioned before, this will allow said programs to perform elevated tasks. However, before these header files can be used a system call table and handler must be set up.

An example of a system call table with their respective IDs and byte offsets can be seen in figure 4.3. The byte offset differs as a pointer is four bytes in length.

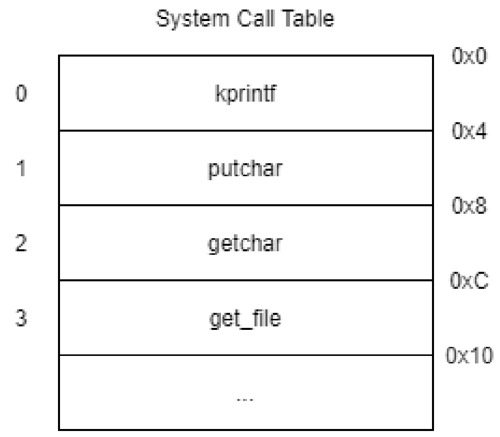


Figure 4.3: An example of a system call table with IDs shown on the left and their respective byte offsets on the right.

A system call ID shall map directly to an index of a handler that is to be called. This section details the implementation of a table and handler that will index said table via a specified system call ID.

sysenter.s

The *sysenter.s* file provides the handler to handle a call from *sysenter*. The definition of this handler can be found in this section. But first a certain assembly function needs to be defined to set the system call handler.

```
__write_msr:
  mov 4(%esp), %ecx
  mov 8(%esp), %eax
  mov 12(%esp), %edx
  wrmsr
  ret
```

The function above will allow for writing to a MSR in the system. The three MSR associated with *sysenter* must be set before it can be used. This function will be used to achieve this later.

The system call handler that will be set is defined below.

```
__syshandler:
  // Preserves the registers.
  // While also adding them as arguments to the handler.
  push %edx
  push %ecx
  push %edi
  push %esi
  push %ebx
  // Gets the system call table.
  lea syscall_table, %edx
  // Generates the correct offset.
  shl $0x2, %eax
  // Indexes with the syscall id.
  add %eax, %edx
  // Calls the syscall.
  call (%edx)
  // Restores the arguments.
  pop %ebx
  pop %esi
  pop %edi
  pop %ecx
  pop %edx
  sti
  // Returns to usermode.
  sysexit
```

Simply put, this handler preserves the GPRs and then uses the system call ID passed through the *eax* register to index into a table of system calls defined by the kernel. As it is known that pointers are four bytes in length, the ID must be shifted to generate the correct offset into the table.

In order to send arguments to a kernel system call, they need to be passed through the available GPRs. Passing arguments through the stack is not possible as the user stack is switched to the kernel variant upon the interrupt, this is shown next. There are three argument registers that are used for system calls *ebx*, *esi* and *edi*. The remaining ones, *edx* and *ecx* are used to store the return address and stack pointer of the caller program respectively. This is beacause the *sysexit* instruction expects these as arguments when returning to user mode. This handler makes sure to perserve them all, while allowing the system call to access them as arguments.

idt.h

Coming back to the IDT header file (see 4.2.3 Interrupts, page 33), a few more defintions are now needed.

```
#define IA32_SYSENTER_CS  0x174
#define IA32_SYSENTER_ESP 0x175
#define IA32_SYSENTER_EIP 0x176
```

These are the ID values of the MSRs that need to be set to the correct arguments for *sysenter* to work.

idt.c

Within the IDT source file, a few calls can be added to set up the *sysenter* instruction.

```
__write_msr(IA32_SYSENTER_CS,    0x8, NULL);
__write_msr(IA32_SYSENTER_ESP,   (uint32_t)&stack_top, NULL);
__write_msr(IA32_SYSENTER_EIP,   (uint32_t)&__syshandler, NULL);
```

The first call sets the code segment selector of the GDT, this must be the kernel. The second sets the kernel stack address, like the one used with the TSS. The final call sets the handler function that will be called by the instruction.

Now that the handler can be called when *sysenter* is used, a table of system calls is needed so that the handler can use a system call ID given by a program to index into the table, finding the correct handler. Below is an example.

```
uint32_t syscall_table[] = {
  (uint32_t)&kprintf,
  // ...
};
```

The table can have as many functions as needed with each function having up to three arguments each.

4.2.7 Filesystem

The file system that this operating system employs is a simple read only TAR archive (see 3.3.1 TAR Archive, page 3.3.1).

Implementation of this system is very simplistic. All programs for the operating system to execute are added into a TAR archive. The TAR archive is then added to the operating system disk image via the linker. Once the operating system is booted, the file system is mapped into memory at boot time by GRUB (see 2.4 GRUB Bootloader, page 16). The interface that is used by the kernel, simply parses this TAR archive.

So because of the ease of parsing, a simple function such as *fs_get_file* can be implemented to retrieve a file with a given path string.

disk.s

Adding the TAR archive to the kernel can be achieved with adding a simple two lines to an assembly file. In this case, a new assembly file called *disk.s* has be declared and added to the build script. These two lines within the file define a new section of the binary called disk and then fill said section with the contents of the TAR archive.

```
.section .disk
.incbin "disk.tar"
```

linker.ld

Circling back to the linker script (see 4.1.2 Compilation, page 22). The address of the file system can be exposed by the linker by adding the disk section to the linker script and defining a linker variable right before. This variables address can then be accessed by the kernel code to retrieve the position of the file system. The lines that are added to the linker script are displayed below.

```
disk_start = .;
*(.disk)
```

filesystem.h

disk_start then needs to be declared as external by the kernel, so the linker knows to map the variable to that region. This is done using the *extern* keyword before the declaration of the variable within a header file. In this case it is *filesystem.h*.

```
extern uint32_t disk_start;
```

Next the TAR file header must be defined. This will be used to extract information about files and folders stored within the TAR file.

```
typedef struct _posix_header
{
  char name[100];
  char mode[8];
  char uid[8];
  char gid[8];
  char size[12];
  char mtime[12];
  char chksum[8];
  char typeflag;
  char linkname[100];
  char magic[6];
  char version[2];
  char uname[32];
  char gname[32];
  char devmajor[8];
  char devminor[8];
  char prefix[155];
  char padding[12];
} __attribute__((packed)) posix_header;
```

A few definitions that describe the type of file within the TAR archive also need to be defined. The type definition of a file in the TAR header is stored in the typeflag field. These will be used later to check if the the requested path represents a file or directory.

```
#define FILETYPE_REGULAR_FILE      '0'
#define FILETYPE_LINKED_FILE       '1'
#define FILETYPE_SYMBOLIC_LINK     '2'
#define FILETYPE_DIRECTORY         '5'
#define FILETYPE_FIFO_SPECIAL      '6'
```

filesystem.c

Now that the address of the TAR archive is accessable, it can be declared as an array of headers. Due to this, the first function that must be defined is a function that calculates the amount of index increments to the next file header of the archive.

```c
inline uint32_t fs_get_jump_size(posix_header * h)
{
  // Gets the size of the file.
  uint32_t file_size, size = str_to_int(h->size, 8);
  // Checks the filetype.
  if (h->typeflag == FILETYPE_REGULAR_FILE)
  {
    // Calculates the file block size.
    file_size = (size / sizeof(posix_header)) + ((size % sizeof(
    posix_header)) > 0);
    // Jumps over the file.
    return 1 + file_size;
  }
  // Jumps over the header.
  return 1;
}
```

str_to_int is a simple function that converts a string of a number to its numerical value using the given base. The definition of this is not provided as it is a simple implementation. This function will find the amount of jumps to the next file header using the *file_size* string provided by the current header.

Using this, it is now possible to iterate over file headers in the TAR archive. For example, a function to get the header of a certain file, given the path, is as easy as the definition below. It should be noted that *disk_addr* is the base address of the TAR archive as a file header array.

```c
posix_header * fs_get_file(const char * p)
{
  // Iterates through the files in the disk.
  for (uint32_t i = 0; disk_addr[i].name[0] != 0; i +=
    fs_get_jump_size(&disk_addr[i]))
  {
    // Checks if it's a file.
    if (disk_addr[i].typeflag == FILETYPE_REGULAR_FILE)
    {
      // If it's equal to the path then the file exists.
      if (strcmp(disk_addr[i].name, p) == 0)
      {
        // Returns a pointer to the file.
        return &disk_addr[i];
      }
    }
  }
  // ...
}
```

The definition of *strcmp* goes without saying. From this function definition alone it is not hard to build a fully fledged parsing library.

4.2.8 Image Loading

The basic tasks of an image loader is to perform a mapping of an binary program into memory. In order for the image loader to map an image into memory it must first parse the image it is to map to find the sections that need to be in memory. The format that this system will use for its programs is the ELF format (see 2.3 ELF File Format, page 15).

imgloader.h

The ELF file provides a header that will be used to extract certain information about the file. Using the header, the image loader can parse the program file to map it into memory and find the entry point to call.

```
typedef struct _elf_header
{
  uint32_t magic;
  uint8_t class;
  uint8_t data;
  uint8_t version;
  uint8_t osabi;
  uint8_t abiver;
  uint8_t pad[7];
  uint16_t type;
  uint16_t machine;
  uint32_t ver;
  uint32_t entry;
  uint32_t phoff;
  uint32_t shoff;
  uint32_t flags;
  uint16_t ehsize;
  uint16_t phentsize;
  uint16_t phnum;
  uint16_t shentsize;
  uint16_t shnum;
  uint16_t shstrndx;
} __attribute__((packed)) elf_header;
```

The loader can use the *magic* fields magic number value to assists in checks that ensure that the image being loaded is in fact an ELF file. The definition of the ELF magic number is defined below.

```
#define IL_ELF_MAGIC 0x464C457F
```

On top of this, a program header struct must be defined as this is used to parse and find loadable sections of the program file so that they can be loaded into memory.

```
typedef struct _program_header
{
  uint32_t type;
  uint32_t offset;
  uint32_t vaddr;
  uint32_t paddr;
  uint32_t filesz;
  uint32_t flags5;
```

```c
  uint32_t flags6;
  uint32_t flags7;
} __attribute__((packed)) program_header;
```

A loadable segment has the type attribute as 0x1.

```c
#define PT_LOAD 0x1
```

imgloader.c

The image loader accepts an address of the header of an ELF file in the file system. This can be achieved using the *fs_get_file* function defined in the file system. It uses this header to find the file data, which is the next header along. Once it has got the data of the file it can then proceed to parse said file.

Within the ELF file header is a pointer to a program headers array. This information is used by the image loader to find segments that have a type of 0x1, meaning that they are loadable segments. The below function can be used to get this attribute and find the address of the array using the program headers offset.

```c
static inline program_header * il_get_ph(elf_header * pfile)
{
  return (program_header *)(((uintptr_t)pfile) + pfile->phoff);
}
```

This array can then be iterated through to check the type of each segment. When the loader meets a loadable segment, its task is to map the segment into memory marked as ring 3 friendly so it can be executed from user mode. It does this by finding the segment in the file using the offset and then copying the segment into a page that has been requested from the memory manager (see 4.2.2 Memory Management, page 28).

```c
// Gets the program header.
program_header * ppro = il_get_ph(pfile);
// Iterates through the program headers.
for (int i = 0; ppro[i].type != NULL; i++)
{
  // Is this a loadable segment?
  if (ppro[i].type == PT_LOAD)
  {
    // Allocates memory.
    pbase = (uint32_t *)pfalloc_alloc();
    // Moves program into memory.
    memcpy(pbase, ((uintptr_t)pfile) + ppro[i].offset, ppro[i].
    filesz);
  }
}
```

The pages that have been requested from the memory manager must have their usermode bit set and get mapped to the correct base address of the binary, which can be found in the ELF header, or a rebasing would have to be performed to change this. The allocated pages can be manipulated via the *paging_map_page* function defined by the page manager. An example using a small program with the base address of 0x100000 is seen below.

```
paging_map_page(0x100000, paging_virtual_to_physical(pbase),
    PAGE_USER | PAGE_PRESENT | PAGE_RW);
```

For different sections, different flags can be set for the memory regions. For example, read only data sections would omit the *PAGE_RW* flag.

After all loadable segments have been mapped into memory, the entry point address that is described by the ELF header is used to call the entry point of the program. This leads into the next section.

4.2.9 User Mode

As described by the goals of this operating system, programs must be kept seperate from the kernel and each other. Once such way of achieving this is to utilise the different ring levels of the IA-32 architecture (see 2.2.3 Ring Priviliage Levels, page 8).

Since the GDT has already been created, switching between RPLs is very easy. The following assembly snippet takes an address as an argument and exploits iret to initialise a swap to ring 3. It does this by setting the segment registers to the offset (with the bottom 2 bits activated, to signal the ring level, in this case 3) of the user mode descriptors described by the GDT (see 2.2.7 Interrupts, page 13).

```
__r3_execute:
  mov $0x23, %ax
  mov %ax, %ds
  mov %ax, %es
  mov %ax, %fs
  mov %ax, %gs
  mov %esp, %eax
  push $0x23
  push %eax
  pushf
  push $0x1B
  push 4(%eax)
  iret
```

Using this function along with the entry point of the program that has been mapped into memory will start executing the program in user mode.

Chapter 5

Testing

The purpose of this section is to evaluate the final operating system product, in terms of the goals of the project. The following subsections evaluate the operating system agaist each defined aim and provide a breakdown of why and why not it passed and failed.

The operating system will be tested on a popular system architecture emulation software known as QEMU[1]

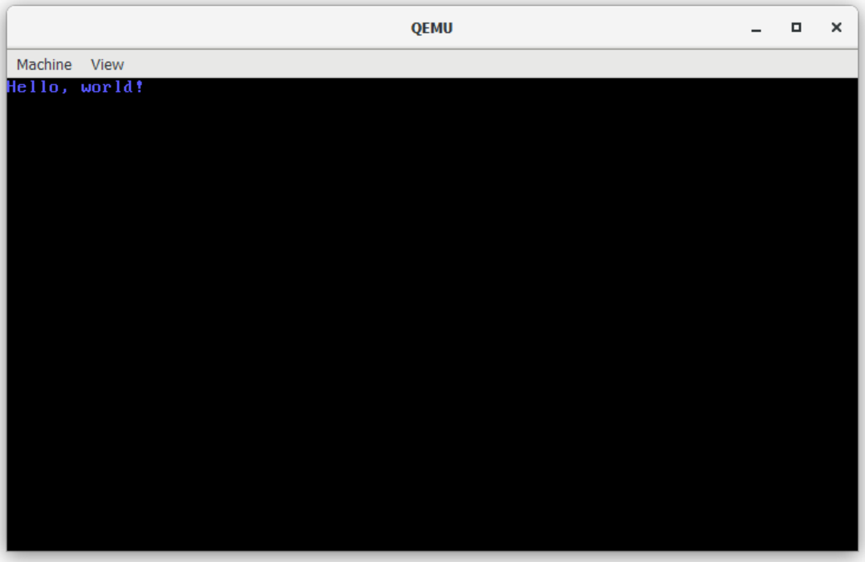
5.0.1 Running Programs

As defined as the main goal of this project, this operating system should provide the ability to run arbitray binaries that use system calls to perfrom elevated tasks. This can be tested by creating a program for the operating system that calls the *printf* system call defined in the project implementation. The program can be compiled into the ELF format using the same command defined for GCC and added to the TAR archive that serves as the operating system file system. The operating system can then read the program from the file system and load the program using its image loader. If all goes to plan, the operating system should output the provided string to the screen.

```c
int main(void)
{
  // Prints the console string to the screen.
  printf("Hello, %s", "world!");
}
```

The program calls the *printf* system call that is defined in the program headers provided by the operating system. The output of this program can been seen in figure 5.1.

[1]More information on QEMU can be found at: https://www.qemu.org/.



```
Hello, world!
```

Figure 5.1: Ouput of the operating system running a ”hello world” program.

As the operating system has produced the expected textual output, it is safe to say that the operating system can in fact navigate the file system, run programs and provide working system calls to service each.

It should also be noted that keyboard input does work as well, however, there is not yet a system call available to read from the buffer.

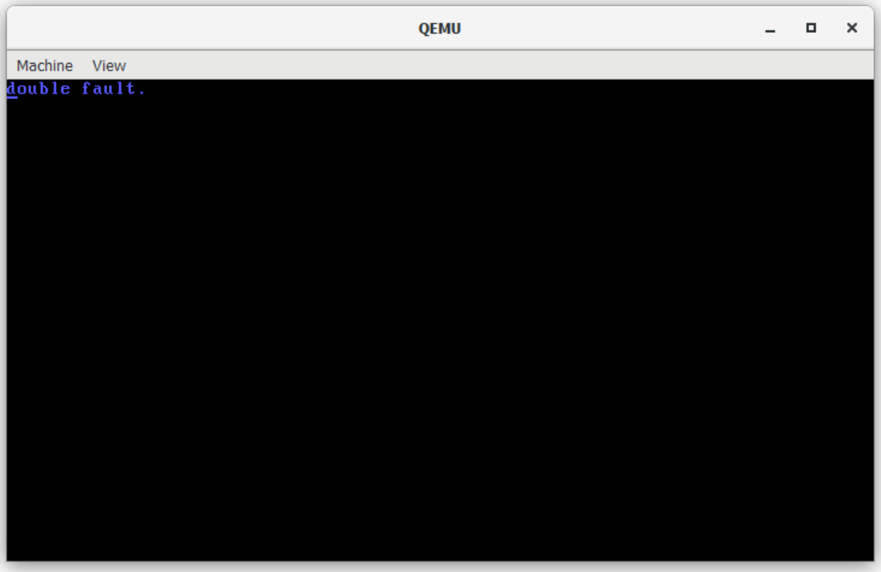
5.0.2 Exception Handling

Another defined goal of the operating system is to provide the same security guarantees as a commerical operating system. This includes exception handing so that the program can not directly halt a system without the kernel catching the exception first.

The program below which will be run on the operating system, triggers a *divide-by-zero* error. This expetion is triggered when the processor attempts to divide a target number by zero. As an exception handler was installed to the IDT, this interrupt should be caught, allowing the kernel to reverse any damage and return to the calling process.

```c
int main(void)
{
  // Divide by zero error.
  printf("Hello, %x", 0 / 0);
}
```

A *divide-by-zero* error can be triggered using *”0 / 0”* in a program that is to be run by the kernel. The figure 5.2 displays the output of the operating system when this error is encountered.



```
double fault.
```

Figure 5.2: Ouput of the operating system running a program with a divide by zero error.

The exception was in fact caught, however it was caught by the double fault handler and not a *divide-by-zero* handler. This is a win in some respects, however, more handlers should be inplemented to deal with certain exceptions. This will make fixing the error and returning to the process the triggered the exception much easier.

Chapter 6

Conclusion

6.0.1 Project Review

This section attempts to evaluate the success of the project by reviewing each individual aim.

1. **Explore the IA-32 architecture and provide a write up of the major features it offers.**

   A large amount of research was conducted in order to find the most notable features of the IA-32 architecture. This information is presented in the Background section, which can be considered the strongest part of this project alone. The informaton listed was found in many materials, such as books, documentation, websites and other academic papers. Exploration of the architecture was found in these materials as well as first hand experiance of being able to work with an IA-32 system.

   The write up that has been provided explores many features of the IA-32 architecture itself, along with general operating system concepts and tools relevant to the project. More diagrams may have sufficed, but the consensus is that there is optimal information to provide a good overview of the architecture at hand. Furthermore, this section suppliments the implementation of the operating system in a structured way and can be easily referred to by a reader if required which is a great benefit.

2. **Compare and contrast different algorithms and implementation methods to provide a fast operating system.**

   The design of the operating system and project itself was considered. However, due to lack of relevancy to the final product, a lot of compare and comparisons were omitted.

   This being a weaker aim of the project itself, was left on the back burner so other, more essential aims could flourish. Fast algorithm implementations were indeed implemented, but a strong argument was not made for their use, merely a write up of their sole benefits.

Testing for speed was not considered, partially again, due to it being a weaker aim and not as feasible as once assumed.

This is a bit of a step back. However, arguments were made for the architecture of the operating system itself and systems implemented by the operating system in general.

3. **Provide a written guide on how to use the IA-32 architecture to set up a basic operating system.**

   The second strongest part of this project, is the Implementation section. This section provides a linear overview and implementation guide of the systems that need to be implemented to provided a working monolithic operating system capable of running arbitrary programs. Although some crucial information may have not been covered, such as building a cross-compiler and using the GRUB multiboot header, the guide created presents a logical step by step process that can be followed to achieve the same outcome of this project. One reason for this could be because these are dependencies that can have multiple implementations, therefore covering just one would present unnecessary information.

4. **Explore the ELF file format to provide the ability to run arbitrary programs in a secure environment.**

   Referring back to the Testing chapter, this was a aim that was indeed met. However, programs are very limited on what they can provide. For example, system calls are lacking for some crucial abilities, such as keyboard input. Although this aim was met, more could have been implemented to add more functionality to programs in general.

   Programs can be added to the file system of the operating system and executed during run time using various functions from within the kernel. Functions were provided to read arbitrary files from the file system and launch them regardless of what they did. It is safe to say that arbitrary programs can be run by the kernel.

   In terms of a secure environment, programs can be run in user space without being able to access kernel memory or functions. This is the ability that was initially expected.

   Moreover, more information on the ELF file format may have been a good addition. Even though sufficient, a more indepth dive into the format of the files may have been more beneficial to the reader.

5. **Provide the same security guarantees that are expected from a commercial operating system.**

   Exception handling was implemented early on within the project. However, this was never built upon. Exceptions such as double faults can be caught by the kernel, but they are not handled. This was expected, as more important aims, such as implementing programs, was considered.

On the one had, the kernel can catch errors generated by a user program, stopping the system from crashing, which was the overall goal of this aim. On the other hand, more could have been done to assist the kernel in reversing any damage and returning control to a user program.

As talked about in the previous aim, memory virtualisation and user space has been implemented. These features stop the process from leaving its secure environment where its is safe from other processes and cannot access memory or instructions that it should not. In terms of security, this is a major implementation and can be considered a reason for why this goal was met.

6.0.2 Future Work

Even though many of the aims can be considered as met, the operating system is still lacking in features. This section takes the time to expand on these potential future features.

Firstly, the kernel provides a very limited set of system calls to user mode programs that need to be expanded on. As a user program is limited in what actions it an perform, more system calls should be implemented to support more functionailty. System calls such as reading keyboard input and navigating the file system are a must for a basic kernel and are simple to implement with all of the systems already in place to do so.

Secondly, systems such as multiprocessing and process management should be considered. A basic round robin schedular to allow for multiple processes to run at a given time would be very beneficial to the operating system itself. This will allow the kernel to run and manage multiple programs at once using multiplexing, allowing for more programs to be run at a fast pace.

Thirdly, there are few implementation issues such as the definition of a kernel stack and the page directory not being changed upon the lanching of a program. These sorts of issues could also be solved using a process manager, however.

Lastly, a more indepth user interface should be provided to the user. The currently exisiting interface can only be used to output text. However, it is more than possible to make it slightly more interactive using keyboard input and such.

6.0.3 Final Remarks

I have always been interested in the more low level areas of computer science, so it was a great delight for me to take on a project of this callibre. Doing so has given me a chance to explore my interest in system architecture, operating systems and the computer science field in general. I have been very fortunate to be able to work with the IA-32 architecture to explore certain concepts and build a basic operating system; something I never thought I'd be able to do before undertaking this.

During the course of this project I have cultivated the ability to research for particular materials on cetain topics and the methods on how to apply these materials in practice. I believe this is an essential skill when it come to the field of computer science. On top of this, digesting this information and then rewriting it in a report has allowed me to gain invaluable experiance in writing academically.

With the aims of the project met, it can officially be declared as complete. However, this does not mean that it was not an insightful journey. Many times throughout tenure of this project I ran into bugs and issues that I didn't think I could overcome, implementations that didn't go as planned, programming mistakes that were overlooked, the list goes on. By experiencing these issues in a time sensitive scenario, I learned how to adapt the project to overcome certain problems. This has only made me a better computer scientist as a whole.

The insight that I have gained from this experince can only go on to assist me with bigger and better things. The craving for creating something new only grows stronger in me now.

Before wrapping up, I want to thank Dr Andrew Scott for his consultation on this project. His suggestions helped me through the many brick walls I encountered along the way. I think I would still be implementing interrupts if this wasn't the case.

I now end this thesis with the hopes that this adventure will lead to bigger and better things...

Bibliography

[1] Andrew S. Tanenbaum. (2008) *Modern Operating Systems (3rd Edition).* Upper Saddle River, N.J.: Pearson Prentice Hall.

[2] Benjamin Roch. *Monolithic kernel vs. Microkernel.* [Online]. Available at: https://fdocuments.in/reader/full/monolithic-kernel-vs-microkernel-kernel-vs-monolithic-kernel-vs-microkernel

[3] Intel. (2016) *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1.* [Online]. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

[4] Intel. (2016) *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture.* [Online]. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf

[5] Intel. (1987) *INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986.* [Online]. Available at: https://datasheetspdf.com/pdf-file/609886/Intel/I386/1

[6] Intel. (1988) *8259A PROGRAMMABLE INTERRUPT CONTROLLER (8259A/8259A-2).* [Online]. Available at: https://pdos.csail.mit.edu/6.828/2005/readings/hardware/8259A.pdf

[7] GNU Project. (2019) *the GNU GRUB manual. The GRand Unified Bootloader, version 2.04, 24 June 2019.*. [Online]. Available at: https://www.gnu.org/software/grub/manual/grub/grub.pdf

[8] Intel. (2016) *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, M-U.* [Online]. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf

[9] TIS Committee. (1995) *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2.* [Online]. Available at:https://refspecs.linuxfoundation.org/elf/elf.pdf

Appendix

Operating System Project Proposal

James Mackenzie

Abstract

The following document is a proposal to create an operating system for the i386 architecture. The i386 architecture is named after the 80386 processor that was developed by Intel and first introduced in 1985; it is a 32-bit version of the x86 instruction set architecture. This document begins by giving a brief overview of the motivation behind the project, then moves into related projects, the methods and materials used to complete the project itself are to follow. The document concludes with a work and evaluation plan.

1. Introduction

A lot of operating systems in this current day have a considerable overhead and redundant features as they must provide support to older programs/systems. Some operating systems will use lesser optimised methods implementing features (e.g. memory allocation, file system, etc) in order to reduce errors or security issues. Because of this, modern operating systems tend to run slower than necessary which can be a problem for developers that wish to provide fast running code without worrying about such things. For example, a bitcoin miner that is run on Windows would be considerably slower than a bitcoin miner that has been developed as a unikernel (a program that runs solely on the central processing unit (CPU) without any operating system overhead). The proposal is to provide a middle ground for developers that wish to have fast running code but not port their program as a unikernel. For this, a barebones operating system would have to be developed that provides many of the functionalities of Windows (e.g. running executable files, providing standard libraries, etc) and implement the most optimal algorithms possible. This will allow developers to compile their program for Windows and without having to change anything (or port the application), run the same executable on a faster operating system. The operating system will differ significantly from Windows internally and only claims to support the loading, mapping and execution of Windows binaries.

2. Related Work

ReactOS is an open source effort to create a lightweight operating system that mimics the Windows architecture. As it is based on NT5, the operating system is neither monolithic nor microlithic, in fact it is a hybrid of both. The operating system is constantly in development and has not reached beta yet, however, only recently has support been added for NT6 (Windows Vista onwards). Because of this, a lot of modern Windows programs cannot operate to full functionality on the operating system. As the project is open source, it is possible to view many undocumented NT5 features within the source code. This may prove to be of use later within the project. The project is not as popular as Windows itself, this may be due to the fact it is not backed by a major company. However, it seems like the project offers little more than a Windows and Linux hybrid, which is a rare need in todays climate.

3. Prerequisites

The aim of this project is to produce a multi-threaded 32-bit operating system for the i386 architecture with a command line interface (CLI) that can interact with various different types of hardware and run very simple multithreaded software. This section details the prerequisites that will be required for the project.

Compiler

Due to many C compilers being platform specific (i.e. only building for the platform they are running on) a roadblock is usually hit when attempting to build an operating system. Firstly, many compilers rely on the fact that the binary that is being built will be run on an operating system, therefore the binary is packed with operating system specific headers – which are not needed. Secondly, many compilers will by default include the standard library, this must also be avoided to reduce any errors in the future. As we are targeting the i386 platform we can compile to a 32bit binary – we do not have to worry about changing this. In order to compile a binary for the i386 without any operating system headers or standard libraries, a cross compiler must be used. Cross compilers for specific architectures can be found online or self-built. GCC provides its source code online along with instructions on how to build the compiler and linker for specific systems. The default GCC compiler with a few tweaked parameters can be used, however, a cross compiler will streamline the building process and reduce any errors we may come across in the future.

Bootloader

Before the kernel entry point is called a bootloader must be used to load and transfer control to the operating system kernel. The most common bootloader, which is used by Linux, is GNU GRUB. GNU GRUB is a multiboot bootloader, meaning that it has been developed to work with a wide variety of operating systems in mind. GNU GRUB makes it easy to load the operating system from the disk via a configuration file, thus it does not need to be manually loaded. GNU GRUB will reduce the work needed to get the operating system up and running. A custom bootloader will allow better control over the boot process and reduce bloat, however these advantages become negligible when considering the time saved from using a premade bootloader.

Emulator

In order to test the operating system on the i386 architecture, without using a physical system, an emulator must be used. Some advantages to using an emulator are limited debugging support and streamlining the toolchain. qEmu, an easily installable open source emulator, provides the i386 architecture and can quickly boot an operating system via command line parameters which makes it easily integrable into a toolchain.

Bochs is another alternative and provides a greater amount of debug information, however, because of this there is a lot of tweaks that must be made before getting setup and running **-** not to mention adding to a toolchain. It may be a better alternative to acquire if any problems occur within the project where the debug information is not easily accessible as it provides a much more in-depth console output and logs.

VMWare Workstation, being the superior of the three, is a cleaner cut option. It is very simple to use and can have an operating system setup in mere minutes – which is why it is widely used. VMWare, due to its constant support and its purchasable nature ensures that it has the best drivers for virtualisation. As it is one of the best alternative to having a physical machine, it will be a great way to test the operating system during evaluation.

4. Design and Methods

The design of the operating system must reflect the overall objective which, in this case, is speed. As the objective is speed, the best approach is to implement a monolithic kernel (The operating system will run solely in kernel mode), as this will reduce the need to trap to kernel mode during program execution - which can be very costly. The monolithic approach defines a high-level interface over computer hardware and is widely accepted as being more efficient than a microkernel approach. However, as the NT architecture defines, Windows is a hybrid system. As Windows processes run in user mode, an effort is to be made to achieve the same. Because of this, this system will take a hybrid approach and employ both microlithic and monolithic standards. Implementing user mode for process execution only.

Unfortunately, there will be many constraints will the system. For example, a lot of Windows features are in fact undocumented. Windows Internals defines a lot of the base internal architecture that Windows employs, so it will be of good use, but cannot be relied upon. Because of this, the system does not aim to provide all the support offered by Windows. The system aims to simply provide an execution environment for simple Windows executables. The system will make use of a similar image loader to Windows and address the need for dynamic link libraries, but the internal architecture will differ significantly.

Furthermore, the operating system must use the most optimal methods of implementation for certain features in terms of the project objective. For example, the memory allocation scheme that the kernel makes use of must match the goals of the operating system. These implementations are proposed in the foreseeable sections.

Interface (Ouput)

As the system is built to be simplistic, a full graphical interface is not needed. A CLI will suffice for the services the system must provide. A CLI does contain limited graphical capabilities which can allow programs to build small interfaces, however this is not enough for a full-fledged Windows based interface. As the system will only provide a CLI, a simple implementation would be to use the VGA buffer to output ASCII characters to the screen. Taking the project a bit further, this can be expanded into a pseudo interface without too much of a performance penalty and will make the system seem more responsive.

Interrupts

The i386 architecture employs the use of a programmable interrupt controller (PIC). However, the PIC is considered to be outdated and has since been replaced by the advanced programmable interrupt controller (APIC) - a modern, more efficient, equivalent. Is it still possible to use the PIC, as it is emulated by the APIC, however it is more restrictive than the APIC. Therefore, the APIC should be used. The Intel x86 datasheets which will be referenced throughout this project (and can be found in the references) for many hardware related tasks, describes the various different commands and possibilities when interacting with the APIC.

User Input

The system will support hardware drivers (binaries loaded by the operating system) that can poll and accept interrupts from their respective devices. These devices will consist of a Keyboard and Mouse. Implementing these drivers will allow the user to interact with the system by typing or using their mouse. Adding support for such things are the basis for programs to be interactable. Said device drivers will be separate binaries and will be loaded by the operating system at run time. These drivers will wait for the APIC to toggle their handlers or will poll their device when the CPU is free. However, this system will be predominantly interrupt driven as multiprocessing is not a priority.

Virtualisation and Paging

By default, the system can virtualise up to 4GB of RAM. There is a modern assumption that in todays age people will typically have 8GB+ of RAM. Because of this, an effort will be made to implement

64bit paging for the 32bit system. Once paging has been implemented, an interrupt handler will be developed to handle page faults, meaning that certain pages of memory can be stored on the hard disk when they are not commonly used. If a page fault does occur, then the page will be mapped back into memory. This will allow the memory of the system to be expanded beyond its limitations.

Memory Allocation Scheme

The buddy allocation scheme, which divides blocks of memory into smaller blocks via a binary tree in order to satisfy different requests, is considered one of the most optimised in terms of speed and one of the easiest schemes to implement. However, it can lead to higher internal memory fragmentation. The slab allocation scheme can reduce internal fragmentation, however, is considered less optimal in terms of speed. As the objective of the project is speed and memory allocation must be the fastest possible (there may still be a speed tradeoff), the buddy allocation system is proposed for this instance.

File System Implementation

The kernel filesystem needs to be as efficient as possible, as reading and writing from the hard disk is costly, a volatile filesystem is proposed. A volatile filesystem resides in the random access memory (RAM) only. It is widely accepted that reading and writing from/to the RAM is more efficient than the hard disk. As there is limited need for disk storage and a large amount of RAM available (at least 4GB) a volatile filesystem is a safer choice for this project.

Process Management

Like every other operating system, this system must manage its processes. A process manager will keep track of the running processes and, if implemented, will allow for multiprocessing. A process manager will also load processes, execute them and ensure that they safely exit. Because of its need to keep track of running processes, efficient algorithms must be used to traverse the process list. A process queue may be more efficient for this task as traversing becomes redundant due to the first in first out (FIFO) dynamic. However, this may change when the task is confronted first-hand. The process manager will act in the kernel and make the switch to user mode when the program is to be executed. Multithreading, much like multiprocessing, need to be managed. The system will make use of the same methods employed by the process manager to manage each individual thread in a single process. As these two managers are close knit, there will be a lot of interaction between the two.

5. Workplan

There are many objectives within the project. For example, system input/output must be implemented, user mode and kernel mode, multi-threading, etc. In order to monitor the progress of the project and reduce the workload a Gantt chart (Figure 1) has been produced detailing the various different milestones that must be reached, this will help guide the project to its goal more smoothly.

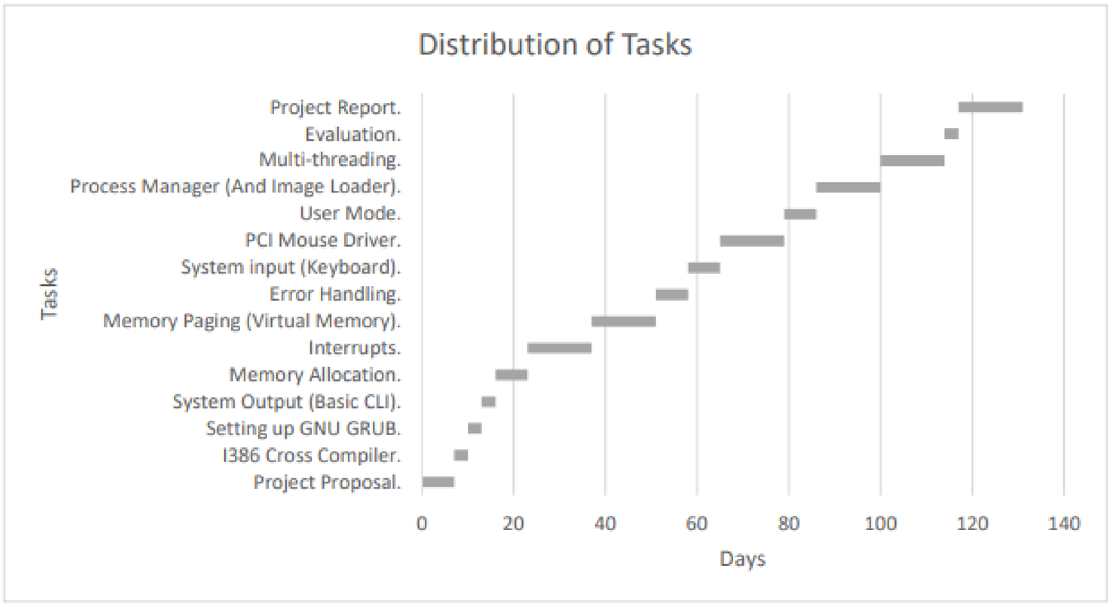


Fig 1. A Gantt chart detailing the distribution of tasks overtime within the project.

6. Evaluation

In order to tell if the operating system has met the overall goal of the project, the final product must be analysed and evaluated. A few questions must be answered:

1. **How scalable is the operating system?**

Can the operating system be built upon? Can it be scaled up/down to work on different architectures? Is it easy to change certain aspects of the operating system without causing issues?

2. **Can the operating system handle user input?**

A Windows like operating system that does not make use of the mouse and keyboard is doomed to fail. Are these devices functional? To what extent?

3. **Can the operating system function and run at sufficient speeds on different systems?**

A key function of an operating system is universality, can it run on different systems? This will involve testing the operating system on different emulator and systems to test if it functions to standard.

To test the speed of the operating system, certain features of the operating system must be benchmark tested. The evaluation method will consist of an internal counter/timer measuring these attributes (manually programmed and output to the CLI), as it may prove to be quite infeasible to run a pre-existing benchmark software on the newly created operating system. The final results will be compared with outsourced information from rival operating systems (such as Windows or Linux).

4. **Can the operating system handle errors? Is it reliable?**

A single error can crash the system, developers will not take too kindly to a system that is not reliable. As Windows provides support for such occurrences, this operating system should do the same. A test consisting of different software on different emulators, that purposely crash, will allow for testing of these error handling features. Does a simple error crash the system or cause a security issue?

5. **Does the operating system provide support for Windows binaries?**

This test will consist of compiling a simplistic binary on Windows and then loading it via this operating system. Can the operating system run it without issues?

6. **Have the overall features of the system been achieved?**

This is a broad question but will allow for evaluation of the features of the operating system. Were they all fully implemented? Does the system function up to standard without them?

References

1. Wikipedia. 2020. IA-32 - Wikipedia. [ONLINE] Available at: https://en.wikipedia.org/wiki/IA-32.
2. ReactOS FAQ - ReactOS Wiki. 2020. ReactOS FAQ - ReactOS Wiki. [ONLINE] Available at: https://reactos.org/wiki/ReactOS_FAQ.
3. Bruce Dubbs - mailto:bruce.dubbs@gmail.com. 2020. GNU GRUB - GNU Project - Free Software Foundation (FSF). [ONLINE] Available at: https://www.gnu.org/software/grub/.
4. Wikipedia. 2020. Monolithic kernel - Wikipedia. [ONLINE] Available at: https://en.wikipedia.org/wiki/Monolithic_kernel.
5. GeeksforGeeks. 2020. Allocating kernel memory (buddy system and slab system) - GeeksforGeeks. [ONLINE] Available at: https://www.geeksforgeeks.org/operating-system-allocating-kernel-memory-buddy-system-slab-system/.
6. Wayback Machine. 2020. Wayback Machine. [ONLINE] Available at: http://web.archive.org/web/20161130153145/http://download.intel.com/design/chipsets/datashts/29056601.pdf.