Development of BenOS: an x86 Operating System

Benjamin McKitterick

MSci. Computer Science w/ Industrial Experience

March 20th, 2020

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted work, I consent to this being stored electronically and copied for assessment purposes, including the School's use of plagiarism detection systems in order to check the integrity of assessed work. I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Date: March 20th, 2020

Name: Benjamin McKitterick

Abstract

The following paper provides an overview of the fundamental concepts forming a general-purpose operating system, focusing specifically towards the x86 CPU architecture. It supplies the groundwork for initialising a build environment using the C programming language and GRUB Multiboot, while provisioning the reader with a strategy for implementing the aforementioned concepts; focusing mainly towards memory management and multitasking features. The implementation of key operating system features will be detailed, and in order to evidence capability of the developed system, code will be tested, and functionality illustrated. By the end of the paper, the reader should have acquired all the necessary knowledge to build their own minimal operating with all the low-level capabilities of a modern OS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An operating system is the fundamental system software that manages computer hardware by abstracting it for application software to interact with and execute programs elegantly, without having to understand awkward and inconsistent hardware interfaces. To put this into perspective, an application request to print a document is designated to the operating system, which then transmits instructions to print via the printer's drivers. Thus, the application has no need to concern itself with such tasks, rather, the operating system handles the low-level details and provides a simpler way for applications to manipulate hardware using common system services, libraries, and application programming interfaces (APIs).

One of the most profound aspects of operating systems is how they vary in design to accomplish different activities. Personal computer (PC) operating systems support multimedia applications and supply an interactive user interface tailored towards an individual. Mainframes, however, are primarily designed to optimise utilisation of resources, while ensuring tasks are executed as efficiently as possible [1]. Computers have been developed for a multitude of reasons, from operating metropolitan public transport systems to configuring settings for something as insignificant as your home refrigerator. In 1965, Gordon Moore observed that the number of transistors on an integrated circuit would double every two years [2]. Computer performance increased while their size was reduced, resulting in this massive diversification of computers and operating systems we have today.

Operating systems have been in development from the 1940s to this present day [1] and have an affluent and impactful history that is still being transcribed. Modern desktop operating systems are capable of handling a large number of tasks at the same time, though this has not always been the case. Early systems like DOS didn't provide multitasking. In fact, it wasn't until the 1990s when Microsoft introduced preemptive multitasking as a core attribute to their systems; during the development of Windows NT 3.1 and Windows 95, that multitasking started to catch on. The Apple Macintosh operating system: Mac OS X, later adopted multitasking for all its native applications, and afterwards, other companies followed [3].

This paper is an exploration of the development of a general-purpose operating system. It will help to clarify rudimentary operating system concepts and will provide the preliminaries for building and running a basic operating system. Core operating system features, in particular: interrupt handling, memory management, and scheduling, will be investigated and explained carefully and in-depth. A comprehensive guide of how to implement a minimalistic operating system with the aforementioned features will then be developed, and the resulting implementation tested. An operating system with the ability to multitask is one of the many practical capabilities a modern operating system should be able to provide. Thus, the project will conclude with a simple kernel capable of running multiple processes seemingly at once.

1.1 Aims

The aims of this project are as follows:

- Provide a synopsis of the attributes that form an operating system.
- Investigate the necessary requirements needed for a basic operating system to function for a specific architecture.
- Review the previously researched requirements and formulate a rough design with justified decisions.
- Produce a procedural guide and implementation of a minimal operating system, including dynamic memory allocation and effective multitasking.
- Examine whether the implemented operating system conforms to the proposed design and discuss how it could be improved in the future.

Chapter 2

Background

An operating system is a conglomerate of various data structures and mechanisms. Each structure is an intricate segment of the system, containing precise and well-defined inputs, outputs, and functions. This section attempts to delineate these concepts and unfold their underlying technicalities, enlightening the reader and giving them a clearer understanding of what an operating system is composed of.

Before delving into these concepts, it is important to grasp an understanding of the foundation and environment needed for an operating system. Consequently, this chapter diverges. The first section discusses the principles of system architecture, system start-up, and build environment set-ups. The subsequent section will examine the major components and services of an operating system and their ability to support the execution of software applications.

2.1 Computer Architecture

Basic computer architecture makes it possible to attain a functioning operating system. It is principal in determining how an operating system is organised, designed and structured. The architecture defines memory organization, exceptions, bus structure, IO and the basic instruction set. Moreover, the majority of the functions performed by the operating system are reliant on the underlying architecture. It exists to define everything at the machine language level and to create an abstraction that manages the machine languages complexity and inelegance.

Processor architectures can be divided into two general classifications: Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) processors. Intel Pentium and Intel x86 are popular CISC type processors and MIPS, Microchip PIC, and ARM are well-known RISC types. Their main differences are that CISC has the capability to execute multistage operations within a single instruction. On the other hand, RISC type processors only use a small set of basic instructions [4]. The proceeding sub-chapter will focus on the CISC based x86 architecture, due to its popularity in general-purpose computer processors [5].

2.1.1 x86 architecture

Adoption of the x86 architecture within general-purpose computing has remained impactful ever since its arrival in 1978 as part of Intel's 8086 chip [6]. The architecture has retained its dominance of the market through its widespread use and perpetual innovation. For that reason, selecting the x86 architecture for a basic operating system will improve the scope for compatibility amongst many current systems. In addition, due to its prolificity, a large

collaborative community is present to give advice and support to those exploring development with this architecture. This paper focuses on the development of a small 32-bit operating system; thus, it is logical to examine a version of x86 that supports 32-bit computing.

Intel Architecture 32 bit (IA-32) is the 32-bit version of the x86 instruction set architecture of which provides "extensive support for operating-system and system-development software" [7]. Support provided for this includes various modes of operation: real mode, protected mode, virtual 8086 mode and system management mode, more commonly known as legacy modes. The architecture provides instruction support for memory management, interrupt and exception handling, task management, and multi-processor control. This class of processors is heavily documented by the *Intel Architectures Software Developer's Manual,* [7] which consists of a comprehensive collection of detailed instructions on how they can be operated.

On a final note, the x86 architecture provides protection domains that constitute all permissions granted to a specific code source. As a result, the processor's built-in protection procedures prevent user-level applications from accessing kernel-level code. This protection is pragmatic for assisting in debugging and detecting software issues during development [7]. Similarly, x86 provides support and protection for memory allocation constructs. The mechanisms provided for segmentation and paging (see section 2.5.2) encourage and support a broader diversification of approaches to memory management. In protected mode, protection mechanisms become operational at both the segment level and page level. Optional flags can be set in the control register that will switch the processors legacy mode to protected mode, enabling segment-protection mechanisms [7].

2.2 Build Environment

Firstly, to make an operating system, an environment for development must be established. A build environment refers to how software is assembled into an executable package. It should contain all the required tools and sources that will allow the toolchain (see section 3.1.2) to complete the tasks necessary to assemble deliverables. To clarify, toolchains are a set of programming tools that permit more advanced software development. A simple toolchain may consist of a linker, libraries, a debugger and a compiler, combining to transform source code into executables.

The issue then raised is how executable code can be compiled for a platform other than the one on which a compiler is running. This problem is solved by a cross-compiler, also known as a retargetable compiler, which runs on a separate platform from the one it is generating an executable for. The host platform would be the current operating system being used for development, and the target platform is the operating system about to be created. When running applications or using shared libraries, a platform must conform to certain regulations. An Application Binary Interface (ABI) defines structures and methods that a compiled application can use to access external libraries, similarly, to an API but at a lower level [8].

This leads us to the System V Application Binary Interface (System V ABI), a specification set for compiled application programs as well as a minor environment that supports installation scripts. It is a standard ABI used by many Unix operating systems like Linux. The Executable

and Linkable Format (ELF) defines the interface between an operating system and an application program, whilst conforming to the System V ABI standards. To explain this more clearly, when an operating system is instructed to run an application, a certain format is expected. It expects the first section of the binary to be an ELF header holding data about memory offsets. This is how the application is able to communicate vital data about itself to the operating system [8].

2.3 Bootloader

To fully comprehend the boot-up process, a little foundational knowledge is initially required. Firstly, the Basic Input Output System (BIOS) is a type of firmware used during start-up; firmware is software that is stored in non-volatile memory. When the computer is booted, the BIOS initialises and attempts to find a boot device in the BIOS settings, such as boot sequence, date and time, and fan speed, all retained in the complementary metal-oxide-semiconductor (CMOS). An initial sector from the boot device, known as sector 0, is loaded into memory and executed. The sector contains a Master Boot Record (MBR) program that inspects a partition table at the end of the boot sector. A bootloader program is then read into memory from the partition.

To elaborate, the MBR is a traditional standard BIOS format that contains a partition table at its end. The table provides the start and end address of each partition. After locating the partition marked as ‘active’, the MBR reads in the first block, known as the boot-block, and executes it. The bootloader in the boot-block then loads the operating system [3]. Newer systems use the GUID Partition Table (GPT) format, forming a part of the Unified Extensible Firmware Interface (UEFI) standard [9]. This format is used in BIOS systems limited by MBR partition tables, which only use 32-bit logical block addressing of the conventional 512-byte disk sectors.

Now that the background of the boot-up process is understood, the function that the bootloader plays can now be perceived clearly. Ultimately, the bootloader is in charge of loading the kernel into memory, supplying the kernel with necessary data, transferring control to the kernel, and swapping to a more suitable environment. The bootloader dictates the load order of files and initialises the first operating system processes. Depending on the architecture, practical data for querying video modes, address space mapping, CPU bit length and Global Descriptor Tables (GDT) (see section 3.2.1) can be obtained for the kernel.

Every type of CPU architecture has its own specific instruction set. Thankfully, tools like GNU assembler provide a single language to communicate to all the various architecture instruction sets, aiding compatibility between architectures [10]. Consequently, this makes it no longer necessary to write a bootloader for each architecture that the operating system desires compatibility with. GNU assembler’s primary purpose is to assemble the output of the GNU C compiler [11] for use by the GNU linker. The linker script combines an ensemble of objects and archive files, relocating their data and linking symbol references [12].

To pack the operating system into a binary image, it is imperative to link the bootloader script with the compiled operating system objects. In 1995, a Multiboot Specification was designed

by Erich and Brian Ford to allow any compliant operating system to be booted by any compliant bootloader. They were adamant not to augment the already existing number of incompatible boot methods, and so GRUB was created [13]. It is a very powerful and flexible bootloader, capable of understanding filesystems and kernel executables. Thus, you can load any operating system without having to record the physical position of the kernel on the disk. As specified by the GRUB manual, additional functions like configuration files, menu interfaces, multiple executable formats and filesystem types are supported [13]. An operating system loaded via GRUB can be emulated and tested using software like QEMU and BOCHS, due to their Multiboot header support.

2.4 Kernel

The kernel is essentially the core of an operating system and the first program loaded after the bootloader. Control is passed over from the boot script to the kernel, which then handles the rest of the start-up operation. A kernel's responsibilities are mainly managing IO requests from software, sending direct instructions to the CPU, handling hardware peripherals like keyboards and speakers, and managing memory. The proceeding content of this sub-chapter will explore the role of the kernel in more depth.

Having full access to all the system hardware, the kernel acts in an elevated system state. It resides in a protected memory space known as kernel-space. To put this into perspective, a user application executes in user-space and is allotted a subset of the systems resources and memory by the kernel. If an application wants to carry out a task, it must do so via system calls; the kernel will then act on behalf of the application [14]. In this sense, the kernel acts as a proxy between the user applications and system resources, providing a layer of abstraction and protection. This layering allows the kernel to control and safely multiplex system resources.

There is a spectrum of kernel designs that have been tried and tested in operating systems. The two most common organisations are monolithic kernels and microkernels [3]. Monolithic kernels are implemented as a single process executing in kernel space. MS-DOS, Solaris a Unix kernel, and Linux a Unix-like kernel are few of the many existing kernels that follow this design. Microkernels, however, separate user applications and kernel system software into different address spaces. They communicate through message passing using an inter-process communication mechanism (IPC). Mac OS X, Minix 3, and Windows NT are examples of such architectures. See figure 1 for an illustration of both kernel structures.

Most Unix systems are monolithic in design and conform to an IEEE standard known as the Portable Operating System Interface (POSIX). The purpose of this standard is to define a system call API that all Unix systems should support, with the objective being to provide compatibility of application software between operating systems [3].

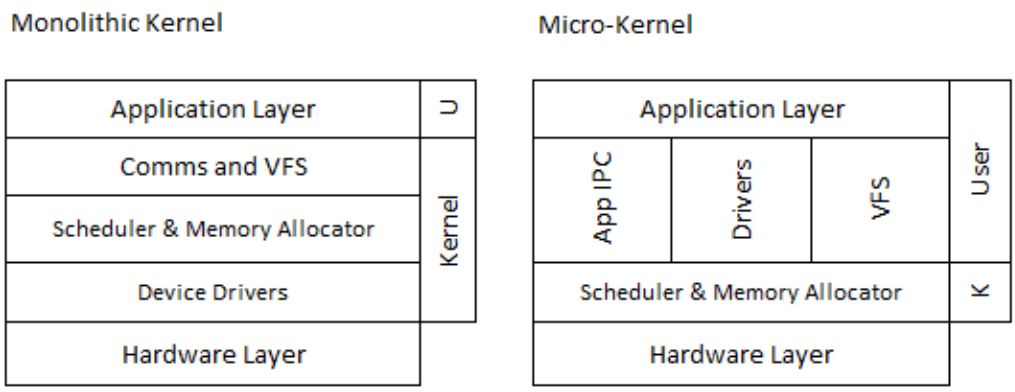


Figure 1 - Common Monolithic and Microkernel organisations.

2.5 Memory Management

The operating system is delegated the task of abstracting a hierarchy of memory into a usable model for system functions and programs to utilise [3]. This is known as memory management and is crucial for developing a fully functioning and effective kernel. As processes execute, they need to be able to store temporary data in order to perform various tasks and calculations. Therefore, the kernel must aptly allocate and deallocate these areas of memory on request from processes requiring it.

2.5.1 Protection and Allocation of Memory

Memory Management Units (MMUs) are computer hardware units that perform the translation of virtual address space to physical address space. This translation prevents process exposure to physical memory, liberating them from having to manage shared memory space. Segregating these areas of memory allows processes to execute without reading or writing in kernel-space or another processes address space. Techniques provided by the MMU, such as arbitrary address space layout and private executable space, supply the system with a way to control memory access rights and protection.

There exist a vast number of implementations for allocating memory [1]. One of the simplest methods is to divide chunks of memory into fixed-size partitions and allocate each partition to a single process. The variable-partition scheme, however, records which parts of memory are free or occupied in a table. Altogether, memory is perceived as a single large pool of memory known as the heap. A heap is allotted to every process and can dynamically grow using *malloc()* and shrink with *free()* [15].

As a final point, a problem known as the ‘dynamic storage allocation problem’ [1], presents the issue of requesting a variety of sized segments from a list of free memory holes. There are a few different algorithms that solve this issue; the most commonly used are first-fit, best-fit and worst-fit. Data shows that in terms of storage consumption and time reduction, best-fit and first-fit are more efficient than the worst-fit strategy [1].

2.5.2 Memory Management Schemes

There are numerous memory management techniques, ranging from very simple, like single contiguous allocation, to greatly sophisticated schemes. This section will focus mainly on two very contrasting approaches to memory management called paging and segmentation.

Segmentation is a memory management scheme that divides logical address space into a collection of contiguous differing sized segments. When a memory location is referenced, to identify it, we use the segment name and offset. The segmentation function retains a segment table that includes metadata about its size and physical address. (see figure 2) Unfortunately, because segmented memory is allocated in irregular chunks, it has the problem of fragmentation. The issue of fragmentation is widely discussed [3] and a sizeable topic, therefore, it will only be covered briefly in this paper. In a nutshell, if memory chunks of differing sizes are continuously allocated and deallocated, eventually small portions of memory begin to build between adjacent chunks. A lot of systems don't use segmentation anymore, but instead employ paging, though a lot of architectures still support it for backwards compatibility [16].

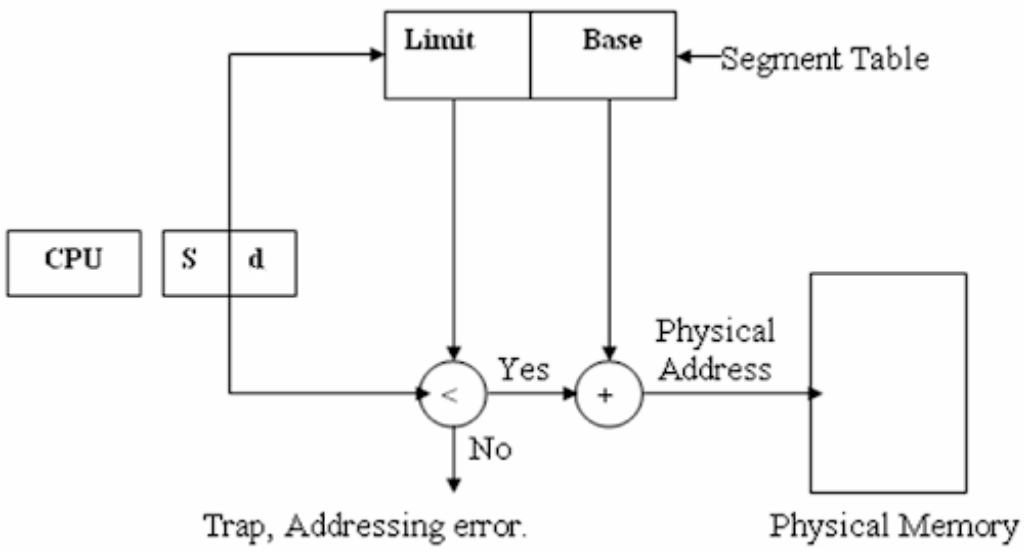


Figure 2 - Illustration of the segmentation process.

On the other hand, paging is a technique that provides an individual mapping of virtual-to-physical address spaces. The address space is divided into fixed-size non-contiguous chunks known as pages, which are mapped to their corresponding physical address counterparts called frames. The mapping is stored in a page table data structure. Paged memory is free from the high level of fragmentation that segmentation suffers from because each chunk of memory is the same size. This scheme makes use of secondary storage to allow processes to surpass the limits of physical memory. Substantially, paging lessens memory wastage, though increases overhead because non-contiguous allocation requires address translation. Admiringly, this overhead can be subdued through a memory cache called the Translation Lookaside Buffer (TLB), which can cache data from paging structures, accelerating the address translation procedure [7].

2.6 Virtual File System

In order to facilitate the interaction between applications and various types of filesystems, Unix-like systems adopt the approach of a Virtual File System (VFS). The VFS is a subsystem of the kernel that converges user-space applications and file systems, providing interoperability between the two (see figure 3). The purpose of the VFS is to create an abstraction between the details of how files are stored, and how they can be accessed. The central focus is to extrapolate the commonalities of a file system into an individual layer that calls the real filesystems to manage the data [3].

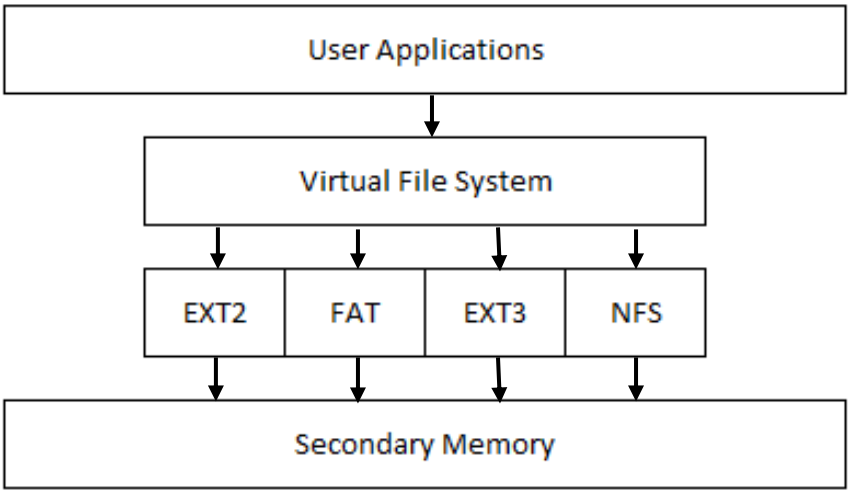


Figure 3 – Abstract representation of the position of the virtual file system.

A file system, according to Robert Love, is "hierarchical storage of data adhering to a specific structure" [14]. Unix filesystems like EXT2, are given a specified mount point known as namespace, allowing mounted filesystems to appear as nodes in a tree. Modern Unix systems endeavour to integrate multiple file systems into a single structure. On the other hand, systems for Windows and DOS split the namespace into segments and allocate them drive letters like 'C:' [14]. Accordingly, when a process opens a file, it now knows which file system to forward the request to via the notable drive letters. This system type makes no effort to coalesce its numerous file systems.

Files are logical groups of data that are manipulated by processes. Their sizes are varied, and so the filesystem must provide mechanisms to handle storing of the non-contiguous blocks representing the file. To solve this, filesystems implement techniques to correctly index, manage and structure these files. The techniques employed are then abstracted by the operating system's virtual file system [3].

2.7 Multitasking

One of the final hurdles in operating system development is multitasking. Once this difficulty is overcome, an operating system becomes far more productive. Multitasking is a system's ability to seemingly execute multiple processes simultaneously, creating the impression that all of them are running at once. The objective is to switch the CPU among processes so frequently that the user can interact with each program while they execute; this concept is known as time-sharing. In this section, basic CPU-scheduling concepts and policies will be covered.

2.7.1 Scheduling concepts

The scheduler is the part of the kernel responsible for delegating processes a 'timeslice' [3], which is the amount of time the process is allowed to execute for. Moreover, it decides what process should run and if a process should cease running. The act of suspending an executing process is known as pre-emption. These pre-emptions in the scheduler are caused by an advanced programmable interrupt controller (APIC) that generates an output signal. The timer will trigger an interrupt every time it reaches a programmed count [7]. When the scheduler receives one of these interrupts, it has to decide if a process should be granted a timeslice and the priority in which to delegate them.

Processes are one of the main pivotal abstractions that an operating system can provide, the abstraction being that it is a running program. A process is an instance of a running program, with additional values of the registers, program counters, scheduling data and variables [3]. When an interrupt occurs, the system needs to reserve the current context of the process that is currently running, in order to restore this context at a later date. Essentially, the process is paused and then instructed to continue later. Switching the CPU to another process involves saving the process state and then restoring the state of another process, this is known as context switching [3].

Traditionally, each one of the previously discussed processes has its own address space and an individual thread of control. A thread is an execution context of all the information a CPU needs to execute a sequence of programmed instructions. The thread contains program counters and registers to keep track of instruction execution order and current working variables. The difference between threads and processes are often misunderstood. Tanenbaum clarifies this misconception by stating "Processes are used to group resources together; threads are the entities scheduled for execution on the CPU" [3]. Threads provide the means for multiple processes to operate in the same environment, independently of one another.

2.7.2 Scheduling Policies

Scheduling policies exist in order to determine the arrangement of queued processes and which ones should be allocated time to execute on the CPU. A vast number of methodologies are

available for CPU scheduling; therefore, this section will cover only two of the more commonly known algorithms, First-Come, First-Served (FCFS) and Round-Robin (RR) scheduling [3].

FCFS scheduling is one of the more simplistic scheduling algorithms. It functions by automatically executing queued requests and process in order of their arrival. This policy can be easily organised by implementing a First-In, First-Out (FIFO) queue. After a process has finished executing and the CPU becomes free, it is allocated whichever process is at the front of the queue. The currently executing process is then deleted from the queue. This algorithm, unfortunately, is not preemptive, thus, not applicable for time-sharing systems. It is vital that time-sharing system processes get a share of the CPU at regular intervals [3].

RR scheduling is specifically designed for time-sharing systems. It operates in a similar fashion to the aforementioned FCFS, though with pre-emption; providing the system with the ability to switch between processes. First, a small segment of time, known as a time quantum, is set from usually ten to one hundred milliseconds. The process queue is then treated as a circular queue that the scheduler traverses over, with each process being allocated the CPU at intervals for the length of a ‘time quantum’. The circular process queue acts similarly to the FIFO, with new processes being added to the back. The scheduler selects the first process from the queue, initialises a timer interrupt for after a time quantum, then assigns the process [3].

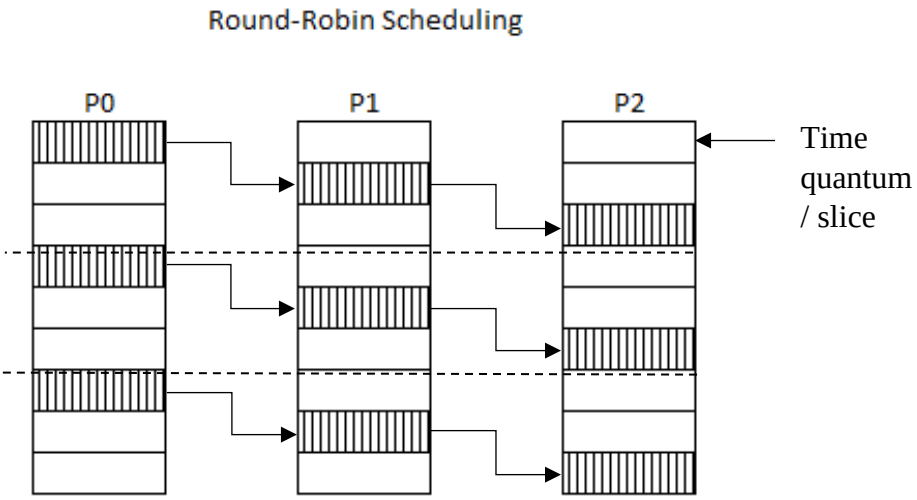


Figure 4 – Round Robin scheduling policy.

Chapter 3

Design

Developing an operating system is complicated and should be planned and researched into great detail before any attempt at implementation is made. The internal modules of an operating system should ideally be orthogonal to one another so that certain operations change one thing without affecting anything else. There should be a clearly distinguished line that separates policies from mechanisms, with mechanisms being handled by the operating system and policies by user processes. The operating system is in charge of abstracting these mechanisms while allowing programs to use them with ease. As an example, a program should be able to dynamically allocate a chunk of memory without having to use functions specific to the computer architecture.

The preceding sections of this report will cover the design of a basic build environment, structures and mechanisms specific to the x86 architecture, drivers for hardware, and provisioning the user with the ability to execute more than one operation through multitasking. There is no absolute path in creating an operating system; the designs presented are a guideline and may not be precisely followed during the implementation.

3.1 Environment

A well-built operating system first begins with securing a strong environment upon which to build. The environment is essential for reducing complexity and unearthing potential bugs later during development. It should be comprised of a set of carefully chosen tools and established in a way that will assist the operating system throughout its development. Taking the time to set up a firm environment now will help in saving a lot of time and aggravation later on.

3.1.2 Tools

The GNU Compiler Collection (GCC) is a compiler system designed to support various programming languages. One of these supported languages is the C Programming Language and will be one of the core tools used to develop this operating system. C is a highly portable general-purpose language capable of byte-level manipulation and direct memory access, with no runtime dependencies.

GNU Binutils is a collection of binary tools and a required dependency for the GCC package. The main tools included in this collection are the GNU linker and the GNU assembler (GAS). A linker is a program that takes one or more object files and combines them into a single executable. This is a necessary tool for working with a language like C, as C compiles individual modules of code into separate object code files. GAS is not a single assembler, but

rather a collection of assemblers for each GCC supported platform. It provides a firm and well-supported assembler to assemble source code into machine code in an object file [10].

To simplify the process of building large binaries from all the different C object files, GNU Make can be employed. It is a tool that controls the generation of executables and will be used to construct the operating system into a bootable image. Moreover, it integrates shell commands and reduces complexity while allowing the user to organise code and customise directory structure. While GCC is platform agnostic, Linux is a preferable environment for operating system development. It is much simpler to create a working toolchain using provided packages such as 'build-essential' to get the tools mentioned above.

3.1.3 Kernel Structure

As you can see from figure 5 below, the directory design here has been largely influenced by the Filesystem Hierarchy Standard (FHS) that defines the directory structure in Linux distributions. Considering that, the developed operating system should be compliant with this standard in order to assist users and software in locating the placement of installed files within the system.

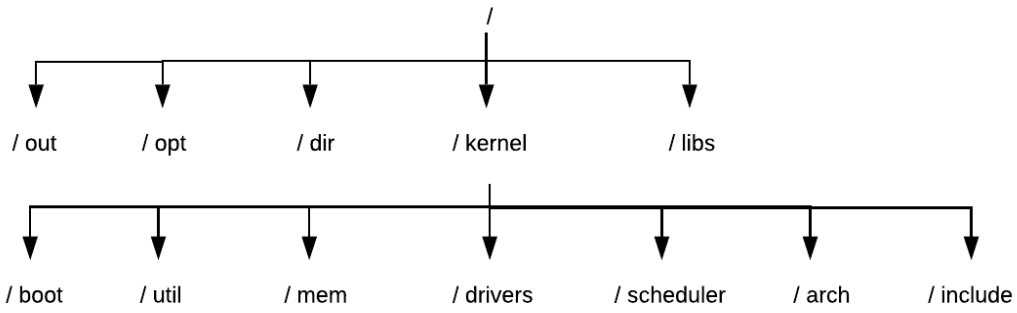


Figure 5 – Kernel directory tree structure

The following are descriptions of the child directories branching from the root node:

/ dir	Built bootable image and config files
/ libs	External helper libraries
/ opt	GCC Cross-Compiler tool set
/ out	All compiled object files
/ kernel	Kernel main and subsystems
/ kernel / boot	Static boot files
/ kernel / util	Utility helper tools
/ kernel / mem	Memory management and allocation
/ kernel / drivers	Device drivers
/ kernel / scheduler	Multitasking and processes
/ kernel / arch	Architecture-specific files
/ kernel / include	Kernel header files

Table 1 – Directories and sub-directories with brief descriptions.

While this paper focuses specifically on the x86 architecture, the kernel structure shown above provides room for modification to allow future support for other architectures if needed. If the developer wanted to provide support for both x86 and ARM processor architectures, then they could simply place whatever data structures and functions that architecture uses, in its own subdirectory; */arm* or */x86*, within the */arch* directory.

3.1.4 Bootloader

Creating and designing a bootloader is a complex project in itself. Instead, it is more appropriate to make the kernel support Multiboot-compliant bootloaders. GRUB implements this specification and allows a kernel to be loaded by including a header called the multiboot header. The Multiboot header is a data structure that should reside in the kernel image to provide information to the bootloader about where and how to load the image and additional multiboot features. GRUB thankfully handles switching the processors legacy mode from real-mode to protected-mode, as well as enabling pin A20 which allows all memory to be accessible [7].

3.2 Structures and Mechanisms

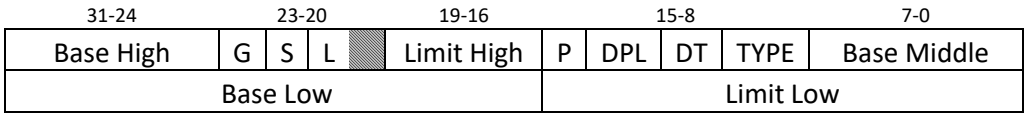
Numerous architecture-specific mechanisms and data structures exist within an operating system. This section aims to cover structures specific to the implementations of x86 architecture. The purpose of these functions and data structures is often to provide abstractions but often come with additional bonuses like protection, organisation and management.

3.2.1 Descriptor Tables

Descriptor tables are native data structures specific to the x86, that offer address protection and management. The structures provide a level of control over the behaviour and characteristics of the processor and memory. Three different descriptor tables are maintained, the Global Descriptor Table (GDT), the Local Descriptor Table (LDT), and the Interrupt Descriptor Table (IDT).

Global Descriptor Table

This descriptive structure possesses the ability to characterise system segments, though it can also hold references to call gates, Task State Segments (TSS), or LDT's. Each segment descriptor entry contains metadata that defines the characteristics of that portion of memory. This includes the physical base address, a limit; the segment size, access rights and two bytes containing various flags for configuration, identified in figure 6.



G = Granularity bit

S = Operation size 16 – 32

L = IA-32e mode only

P = Present in memory

DPL = Privilege 0 – 3

DT = Descriptor type

TYPE = Segment type and protection

Figure 6 – Global Descriptor Tables (GDT) segment descriptor entry.

To reference a segment, a program must load a selector into one of the six-segment registers. A selector is a 16-bit structure composed of: a GDT/LDT (Local Descriptor Table) index, 1-bit identity flag and a 3-bit privilege level. Loading a selector into a segment register allows the processor to read and store the GDT/LDT properties. The 32-bit base address is then combined with an offset to form a linear address, as shown in figure 7 [3].

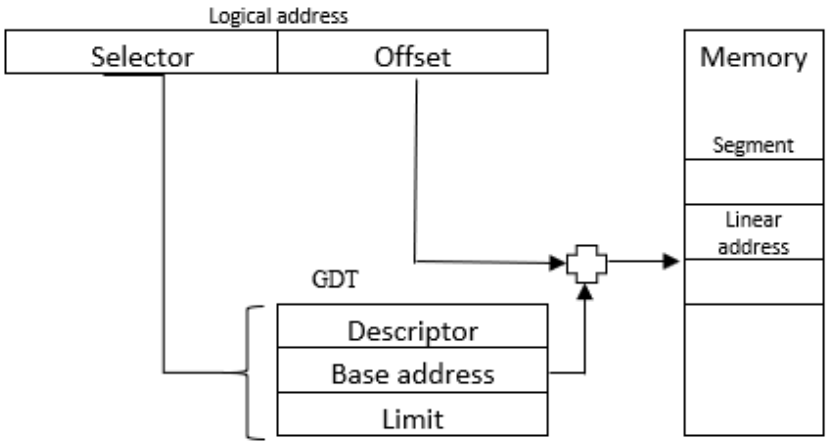


Figure 7 - Transmuting the selector and offset pair into a linear address.

While running in protected mode, memory addresses are managed through either the GDT or LDT. Both structures protect operations run by the operating system, prohibiting accesses to crucial regions of memory.

Interrupt Descriptor Table

Like its counterpart the GDT, the IDT also contains metadata that describes how the processor should respond to various interrupts and exceptions, information on access permissions, and which interrupts are enabled. The IDT holds an array of addresses that correspond to interrupt routines as well as selectors to the GDT and LDT tables.

31 - 16	15 - 8				7 - 0
Base high	P	DPL	S	Type	Zeros
Selector	Base low				

P = Interrupt is present S = Storage segment

DPL = Privilege level Type = IDT gate type

Figure 8 – Interrupt Descriptor Table (IDT) interrupt descriptor entry.

To initialise an IDT, a reference point is formed through a base address pointing to the list of interrupts in combination with the total number of interrupt routines available. The routines referenced, range from peripheral interrupts like the keyboard or Programmable Interval Timer (PIT) (see section 3.3.3), system critical interrupts such as double faults or page faults, and task interrupts.

3.2.2 Interrupt Service Routines

The x86 architecture is driven by interrupts; external event triggers, that intercept and stop any active process from calling an Interrupt Service Routine (ISR). These external events can be triggered through both hardware and software. A typical example of a hardware-driven interrupt would be a keypress which triggers an Interrupt Request (IRQ1), whereas a software-driven interrupt would be activated by the *int* operational code.

To elaborate, a Programmable Interrupt Controller (PIC) controls the CPU's interrupts by accepting multiple IRQ's and communicating these in order to the processor. When a keyboard registers a keypress, a pulse is sent along an interrupt line (IRQ1) to the PIC. The IRQ is then converted to a system interrupt. For the system to know which ISR to call when an interrupt occurs, pointers to the ISR are stored in the previously mentioned IDT's. The IDT's reduce complexity by providing an interface for the interrupts, simplifying the handling of them later. The routine would then pass a value that identifies itself to a more dynamic common handler. This dynamic handler function must end with the *iret* operation code; otherwise, a triple fault could occur. Once the routine has been serviced, the previously paused process can resume its operations.

0x00	ds	0x24	int_num
0x04	edi	0x28	error_code
0x08	esi	0x2C	eip
0x0C	ebp	0x30	cs
0x10	esp	0x34	eflags
0x14	ebx	0x38	user_esp
0x18	edx	0x3C	ss
0x1C	ecx		
0x20	eax		

Figure 9 – Register contents taken from the stack during an Interrupt Service Routine (ISR)

The aforementioned handler function pushes the contents of the stack shown in figure 9 and saves them, to provide the system with additional information about the interrupt. The saved data contains the contents of the general-purpose registers, along with the interrupt number and an error code, allowing the handler to regulate errors itself.

3.2.3 Memory Management

One of the principal features of operating systems is to provide a way to dynamically allocate and deallocate memory. This involves providing a way to reference virtual memory space and translating it to a physical one. As virtual memory is conceptual, it requires concretion through some mechanism or algorithm. Segmentation and Paging are both well-founded solutions for implementing virtual memory.

Segmentation allows physical address space to be non-contiguous. Paging provides this advantage and additionally avoids external fragmentation. Implementing paging requires physical memory to be divided into fixed-sized contiguous chunks called frames and virtual memory to be divided into fixed-sized non-contiguous chunks called pages. This requires the use of three fundamental structures: a page directory, a page table, and a page, shown in figure 10.

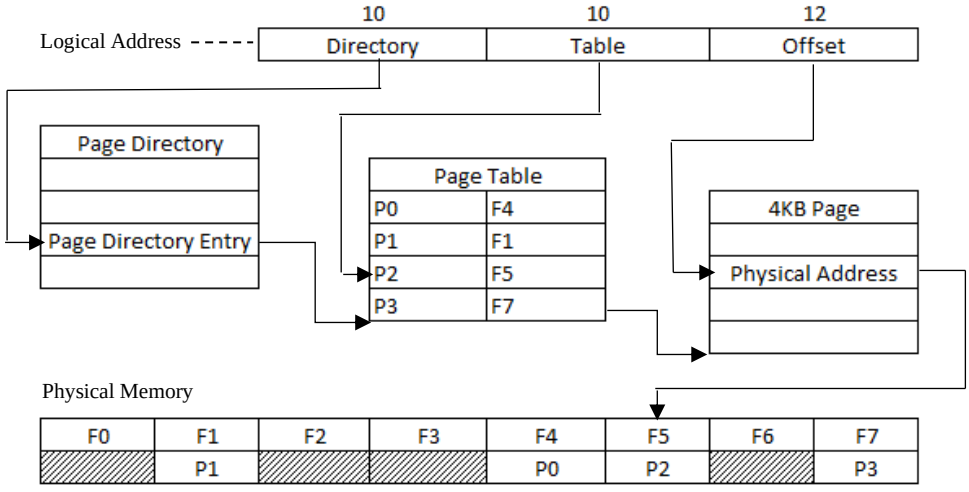


Figure 10 – Logical address translation of a 4KB page using a paging model of logical and physical memory.

A page table has 1024 entries, each containing a 4KB aligned physical address to a chunk of memory mapped to that location. The pages contain numerous flags that determine features such as whether it is present, which mode it is in, and other useful mechanics. The page structure and all its descriptive bits are illustrated in figure 11.

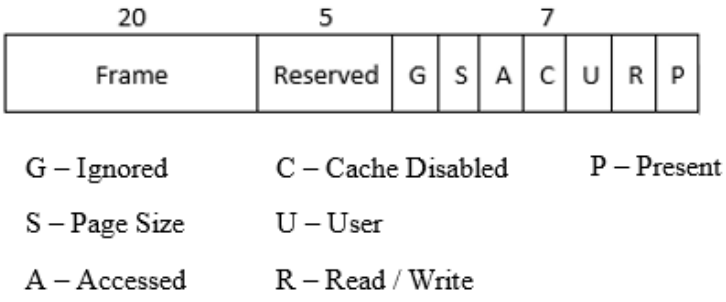


Figure 11 – A page table entry data structure.

An effective way to allocate frames is to use a large bit map that identifies which frames in memory are being used. It is an effective and compact storage technique for holding arbitrary bits; more efficient than merely holding an array of 1's and 0's as it uses 32 times less space. Additionally, bit maps have the ability to exploit bit-level parallelism and limit memory access. They do have disadvantages; however, such as accessing individual bits being expensive and complicated, depending on the search algorithm used. The bits would need functions to set, test, search and clear.

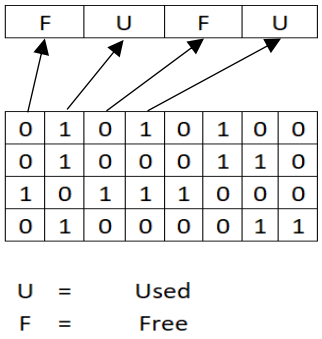


Figure 12 – Physical page frame allocation using a bitmap.

As mentioned earlier, it is vital that an operating system has the ability to allocate and deallocate memory effectively. A data structure known as the heap can provide this; it is simply a structure for keeping track of memory. The heap holds static data and one big free region; known as a hole, which can grow and shrink. When the heap's available space is used, the operating system can request to increase the size of the heap, which moves the heap boundary up closer to the stack. The request to increase size comes from the *malloc()* function and the *free()* function is used to decrease the size and free up memory.

The variable-partition technique involves maintaining a table that indicates which parts of memory are free or occupied. The algorithm explained below uses this technique and is similar to Doug Lea's memory allocator algorithm used in the GNU C library [17]. It is relatively simple to implement, which makes it a great choice for the implementation of a small operating system. To set the picture, the framework of this algorithm must first be explained.

Firstly, blocks are contiguous areas of memory that contain data. Holes, on the other hand, are like blocks, but their memory is not being used. As you can see in figure 13 below, for every hole there is a related descriptor in an index table ordered by hole size. Each of these holes has its own header and footer that holds metadata, the header contains information about the hole, and the footer contains a pointer back to the header, allowing the unification of holes during the freeing process. For error checking, both the header and footer have a magic number field in order for them to stand out and be easily located.

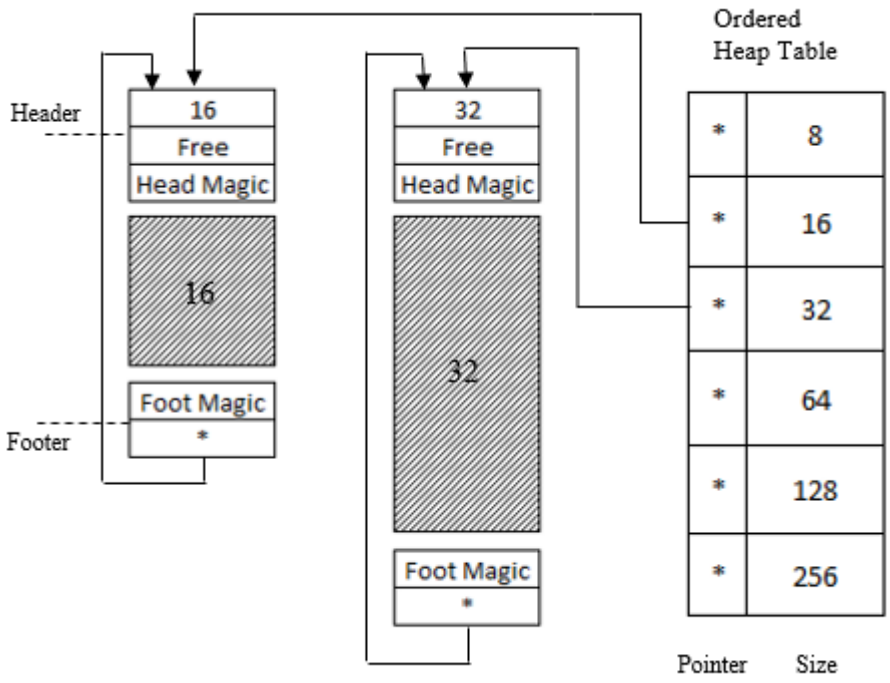


Figure 13 – Ordered heap table point to memory holes and blocks.

This algorithm uses the best-fit strategy, which is advantageous in reducing time-usage and storage utilization [3]. It operates by simply allocating the first hole that is big enough; the heap table is searched through until a free hole big enough for the requested space is available.

3.3 Drivers

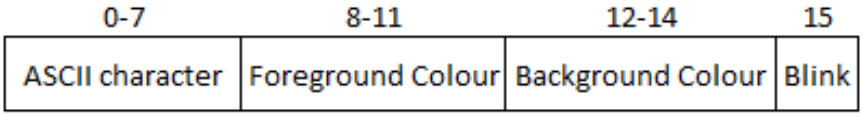
To explain simply, a driver is a piece of software that perceives or participates in the communication between an operating system and device hardware. To communicate, they send data through the computer bus or communications subsystem that the hardware is connected to. They are essential to any functioning operating system; without them, the kernel would be unable to interface with any hardware. Drivers are either built into the kernel or loaded in from external files. The preceding sections of this sub-chapter will cover various built-in drivers that are needed for a minimal operating system.

3.3.1 Display

First of all, display drivers are completely essential to showing system information and will be one of the first things to approach when beginning operating system development. Even though

Video Graphics Array (VGA) is old, it is compatible with most modern graphics cards and emulators like BOCHS and QEMU.

VGA operates with different modes of display that consist of a combination of aspect ratio, display resolution, colour depth and refresh rate. The VGA text mode is a basic way to print a character to the screen; this is done by writing the character to the text buffer of the VGA hardware. The text buffer is a two-dimensional array with customarily 80 columns and 25 rows, of which are directly rendered to the screen. Each entry into this array describes a single character using the format represented in figure 14. It can be located at the physical address of 0xB8000 for x86 processes operating in real-mode.



0x0	Black	0x4	Red	0x8	Dark Grey	0xC	Light Red
0x1	Blue	0x5	Magenta	0x9	Light Blue	0xD	Pink
0x2	Green	0x6	Brown	0xA	Light Green	0xE	Yellow
0x3	Cyan	0x7	Light Grey	0xB	Light Cyan	0xF	White

Figure 14 – character display bit formation for the VGA text buffer.

3.3.2 Keyboard

Standardised methods exist, for providing keyboards with the ability to communicate with a computer. These methods include PS/2, USB and more recently, Bluetooth. Data comes into the computer in a binary stream containing keycodes. It is the keyboard drivers' job to convert the raw keycodes sent by the keyboard and format them into an understandable form.

Port 0x60 can be read to receive data from a PS/2 device. So, when IRQ1 is triggered, the port is read, and afterwards, an End of Interrupt (EOI) with command code 0x20, should be sent to the interrupt controller. The received scan code can then be converted to its ASCII representation using a keymap. Ideally, this is then outputted to the screen via the aforementioned graphical drivers.

3.3.3 Programmable Interval Timer (PIT)

PIT chips are designed for microprocessors to perform timing and counting functions with three 16-bit registers. Each counter has two input pins; clock and gate, and one pin for the output. A 16-bit count is loaded into its register, and then on command, it begins decrementing the count until 0, then a pulse is generated that can be used to interrupt the CPU. The chip consists of an oscillator, pre-scaler and three separate frequency channels.

The oscillator used by the PIT chip runs at 1.193182 MHz, this would be identified as the tick rate. Channel 0 of the PIT is linked with the PIC chip, allowing it to generate an interrupt request (IRQ0). These features can be used to create an infinite series of timer ticks at a chosen

frequency. The PIT also contains channel 1; for refreshing RAM, and channel 2; for controlling the PC speaker, though these are unnecessary for a minimal operating system. PIT channels are registered to I/O ports; specifically, channel 0 is controlled with port 0x40 and the command register with port 0x43.

3.4 Multitasking

Scheduling algorithms can be forked into two divisions with respect to how they handle timer interrupts; preemptive and non-preemptive. A non-preemptive scheduler picks a process, and that process holds the CPU until it voluntarily releases. On the contrary, preemptive scheduling picks a process and allows it to hold the CPU for a set interval. If the process is still executing after that interval has transpired, the process is suspended, and another is picked for execution; as shown in figure 15. For interactive operating systems, users need a responsive environment. Pre-emption is key to stopping processes from monopolizing the CPU and denying its services to others. Moreover, it can prevent processes from blocking others out indefinitely, either from continuously running or through program errors.

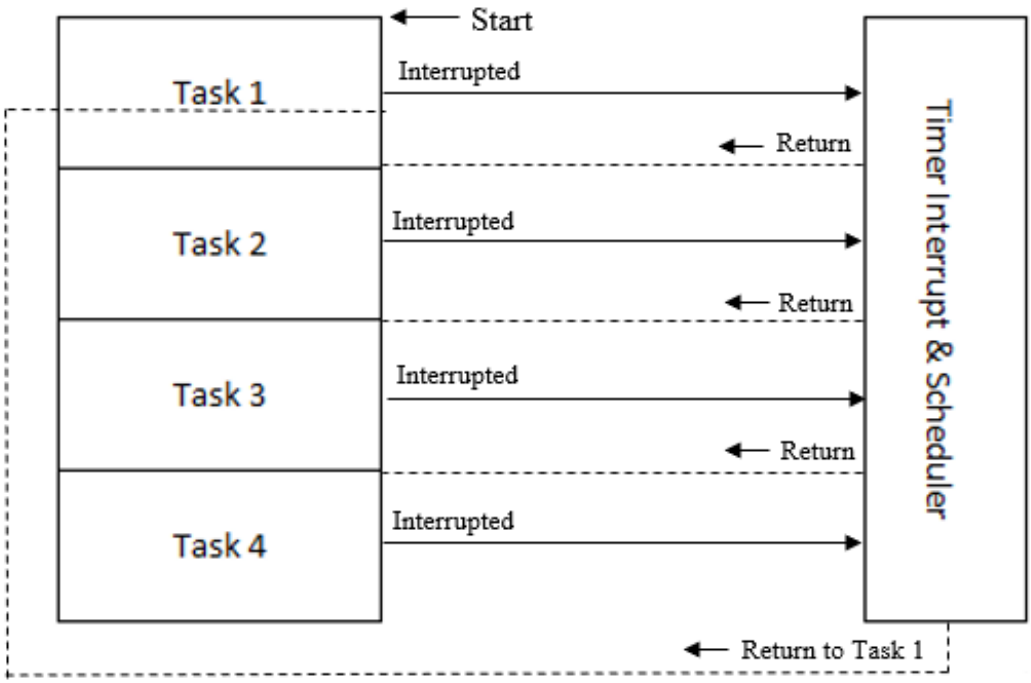


Figure 15 – Preemptive scheduler controlling the task execution sequence.

It is important to correctly understand how processes are modelled in order to implement them later. As discussed earlier in chapter 2.7.1, processes are an instance of an executing program, with their own address space, program counter, registers and variables. It is easier to conceptualise processes as a collective, running in pseudo-parallel, rather than keeping track of how the CPU switches between them.

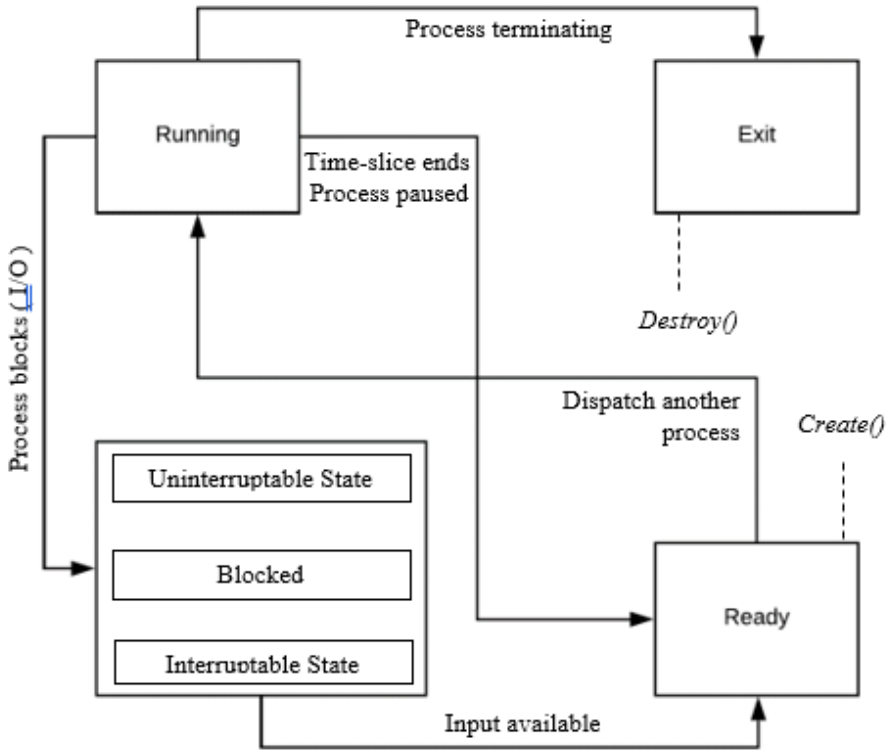


Figure 16 – A process state model showing running, ready, blocked and exiting states as well as the transitions between.

When a process is interrupted, an interrupt service procedure starts by saving the register contents using the same register structure shown in figure 9. The data is pushed onto the stack by the interrupt and stored; the stack pointer is then changed to point to a processes temporary stack. This must be written in an assembly language procedure as high-level languages like C are incapable of these actions. A routine is then called in C to handle the interrupt type. Afterwards, another assembly procedure should start up the new process by loading the registers and memory map, with paging enabled, CR3 should be loaded with the page directory.

Having the process queued in a round-robin is easy to implement, and starvation-free; in that, it will not perpetually deny processes from accessing the CPU. A simple yet dynamic function can be implemented to add tasks to the end of a queue or list. The list can be iterated over and modified during the iteration, to point each task to the next, every time a new task is added. In the end, the tail can be pointed back to the head of the queue to recurse.

Chapter 4

Implementation

This section aims to serve as a rough example of how to implement a minimal x86 operating system. A detailed overview of the development of key operating system features will be covered, focusing mainly towards memory management, memory allocation and multitasking.

4.1 Environment Setup

Firstly, the environment set up is a long, arduous procedure and the examples shown in the preceding sub-chapter are not exact. There are many dependencies and setups that must be performed, and it would take too long to list and explain them all.

4.1.1 Preparation

Before beginning development of the toolset used to run on the host, certain software installations and their dependencies are required. The software can be installed through a package manager like the Advanced Packaging Tool (APT) using *sudo apt-get*:

- *$ sudo apt-get install build-essentials*
- *$ sudo apt-get install qemu*

This will install GCC, GNU Make and QEMU on your system. Compatible versions of Binutils and GCC should be configured and installed on the system used for development. The target and prefix must be specified during installation; this can be done as follows:

```
cd $HOME/src
mkdir binutils-build
cd binutils-build
../binutils.x.y.z/configure --target=i686-elf --prefix=$HOME/opt/cross
make
make install
```

Both GCC and Binutils should be installed similarly to the format above for the cross compiler.

4.1.2 Building a Cross-Compiler

The first step to compiling the operating system is to set up a GCC Cross-Compiler for a generic target called i686-elf. The compiler should be set up locally at *$HOME/opt/cross* rather than installing it into system directories. To add the new compiler to the shell session, add *$HOME/opt/cross/bin* to your environmental variables path. The GCC executable installed at *$HOME/opt/cross/bin/i686-elf-gcc* can then be used to create programs for the i686-elf target. The target provides a toolset aimed towards the System V ABI which allows an easier set up for booting the kernel using GRUB and Multiboot.

4.1.4 Build Automation

Now that GNU make has been installed, the build process for the kernel can be automated by creating a Makefile. This will be used to compile objects from the kernel code, link them and then build a bootable .iso disk image. All of this can be done without having to execute shell commands individually.

The first step of creating a Makefile is to organise any essential directory and compiler argument variables that are required for the build. This will allow better control over the kernel structure and will make the build process much straightforward to read and understand. The code below shows how variables can be used to specify input flag parameters and file paths for directories:

```
# Directories
SRC_DIR ?= $(CURDIR)/src
OUT_DIR ?= $(CURDIR)/out

# Sub-directories
BOOT_DIR ?= $(SRC_DIR)/boot
ARCH_DIR ?= $(SRC_DIR)/arch

# Flags
CFLAGS = -std=gnu99 -ffreestanding -O2 -Wall -Wextra
LFLAGS = -ffreestanding -O2 -nostdlib

# Qemu
QFLAGS = -cdrom
QEMU = /usr/bin/qemu-system-i386
```

The GCC target can then be used within the make file to compile objects. In order to use the target within the Makefile, "*export TARGET=i686-elf*" can be copied into the *~/.bashrc* file; specific to Linux, to set it as an environmental variable. The file is a shell script that runs every time a new shell is opened.

```
# Compile objects
$$TARGET-gcc -I $(INC_DIR) -c $(SRC_DIR)/kernel.c -o $(OUT_DIR)/kernel.o $(CFLAGS)
$$TARGET-gcc -I $(INC_DIR) -c $(ARCH_DIR)/descriptors.c -o $(OUT_DIR)/descriptors.o $(CFLAGS)

$$TARGET-gcc -I $(INC_DIR) -c $(UTI_DIR)/display.c -o $(OUT_DIR)/display.o $(CFLAGS)
$$TARGET-gcc -I $(INC_DIR) -c $(UTI_DIR)/utility.c -o $(OUT_DIR)/utility.o $(CFLAGS)

$$TARGET-gcc -I $(INC_DIR) -c $(INT_DIR)/interrupts.c -o $(OUT_DIR)/interrupts.o $(CFLAGS)
$$TARGET-gcc -I $(INC_DIR) -c $(INT_DIR)/flush.s -o $(OUT_DIR)/flush.o $(CFLAGS)
$$TARGET-gcc -I $(INC_DIR) -c $(INT_DIR)/isr.s -o $(OUT_DIR)/isr.o $(CFLAGS)
```

Each source file is compiled at the destination specified after the -c flag and the object file is placed at the destination specified after the -o flag. The -I flag instructs the compiler where to look for the header files to be included.

Once the files are compiled, the objects are linked using a linker script specified by the -T flag in the Makefile and then built into a bootable image. Make "phony" targets allow functions to be performed by running the name of the target after the make command in the shell. For example, running *'$ make clean*' will invoke a script to remove all build files in the output directory. The make file in this implementation has three individual targets that can be called: all, test and clean. The 'all' target compiles, links and builds. The 'test' target runs the operating system image using the QEMU emulator for testing.

4.2 Kernel Development

4.2.1 Boot loading

To boot the operating system, this implementation uses three main input files: *boot.s*, *kernel.c* and *linker.ld*. The linker file is for linking the boot script and kernel together, and the boot file sets the kernel entry point, while also initialising a minimal environment to pass to the kernel.

The Multiboot specification provides a simple interface between the bootloader and the operating system kernel. Anything extra that needs to be loaded; such as an initial ramdisk for a file system, can be added in the boot script, as the standard allows external modules to be loaded into memory. A magic number should be set and multiboot enabled so that the bootloader can locate the starting point of the kernel. The example below shows how flags and constants can be set for multiboot:

```
.set MAGIC,     0x1BADB002
.set FLAGS,     0
.set CHECKSUM,  -(MAGIC + FLAGS)

/* Enable multiboot and define each of the previously set
   variables to type long. */

.section .multiboot
.long MAGIC
.long FLAGS
.long CHECKSUM
```

To develop in a high-level language like C, a stack is essential and must be initialised. The bottom of the stack should be set first, then stack size allocated, and afterwards the stack top. The *esp* register should be loaded with the pointer to the top of the stack because on x86 architectures the stack grows down. Once the pointer is set, the main kernel file can be called.

The two compiled boot script and kernel objects each contain part of the kernel and must now be linked together to create the full kernel. The location of various object file sections must be defined and placed into the final kernel image; this must be provided in an external customised link script like the *linker.ld* file shown below:

```
5   ENTRY(load)
6
7   SECTIONS
8   {
9           /* Put sections at 1 MiB. */
10          . = 1M;
11
12          /* code section. */
13          .text 0x100000 : ALIGN(4K)
14          {
15      text = .; _text = .; __text = .;
16                  *(.multiboot)
17                  *(.text)
18          }
20          /* Read-write data section. */
21          .data BLOCK(4K) : ALIGN(4K)
22          {
23      data = .; _data = .; __data = .;
24                  *(.data)
25          }
26
27          /* Read-write data and stack bss
28          .bss BLOCK(4K) : ALIGN(4K)
29          {
30      bss = .; _bss = .; __bss = .;
31                  *(COMMON)
32                  *(.bss)
33          }
```

Line 5 tells the bootloader where the kernel entry point is, then the script specifies where the various object file sections should be placed in the kernel image. It is conventional to place the sections at a physical address space above 1Mb to avoid disrupting other BIOS address mappings. A final line at the end of this code stores the end of the multiboot sector in a variable for the kernel to know the starting address of free space to allocate memory:

```
end = .; _end = .; __end = .;
```

Now all the necessary components are available to link the system objects and build a bootable kernel image. The *grub-mkrescue* program can be used to create a bootable CD-ROM image that contains the GRUB bootloader and system kernel [13]. A basic *grub.cfg* file can be used to add extra configurations like customised menu entries:

```
menuentry "benos" {
        multiboot /boot/benos.bin
}
```

Once again, GNU Make can be used to automate building the image. A directory is made to hold all the necessary files for configuration and image building; then the *grub-mkrescue* program is instructed to build it into a *.iso*:

```
# Build the bootable image
mkdir -p dir/boot/grub
cp $(BOOT_DIR)/benos.bin dir/boot/benos.bin
cp grub.cfg dir/boot/grub/grub.cfg
grub-mkrescue -o benos.iso dir
```

4.2.2 Descriptors

Now that the source files can be easily formed into a binary image, actual kernel development can begin. The first thing that should be done is to initialise the architecture-specific descriptor tables. Specifically, the GDT should be the first descriptor table to be initialised in the kernel so that privilege levels and memory regions can be defined. The following extract of code shows how to set up a list of segment descriptors that form the GDT:

```
static void gdt_init()
{
        /* Sets the gdt pointer to allow the processor to find it. */
        gdt_pointer.limit = (sizeof(gdt) * GDT_NUM) - 1;
        gdt_pointer.base = (uint32_t)&gdt;


        gdt_setup(0, 0, 0, 0, 0);
        gdt_setup(1, 0, 0xFFFFFFFF, 0x9A, 0xCF); // code segment
        gdt_setup(2, 0, 0xFFFFFFFF, 0x92, 0xCF); // data segment

        load_gdt();
}
```

To initialise the GDT structure, a pointer base and limit is defined from the size of the struct and number of GDT setup entries, allowing the processor to locate it. A null descriptor should be set first, then the code segment and data segment descriptors; as is expected by all x86 architectures. Any other segments like user-level, LDT's or TSS, can be added after if needed. The *gdt_setup()* function is used to perform the necessary bitwise operations to configure the descriptor like the representation shown in figure 6, using the C *gdt* struct below:

```
struct gdt {
        uint16_t limit_low;
        uint16_t base_low;
        uint8_t base_middle;
        uint8_t access;
        uint8_t granularity;
        uint8_t base_high;
} __attribute__ ((aligned (4)));
```

```
struct gdt_pointer {
        uint16_t limit;
        uint32_t base;
} __attribute__ ((packed));
```

The GDT struct is '*attribute aligned'* to specify that each segment must use a data alignment of 4 bytes. This forces the table to be aligned in a specific order within memory. When all the segment descriptors have been initialised, an assembly *load_gdt()* function is called. This flushes the *gdt_pointer* into the CPU using the *lgdt* operational code. The *gdt_pointer* struct shown above is '*attribute packed'* to force each structure field to have the smallest possible alignment.

4.2.3 Interrupt handling

Before interrupts can be handled, an entry for each interrupt and its corresponding interrupt number should be added to the IDT. If an interrupt occurs and there is no IDT entry for it, then the CPU will triple fault and reset. The IDT is initialised in a similar manner to the GDT, so once again there an *idt_setup()* function to perform necessary bitwise operations; different from the GDT operations. A unique method for each ISR is parsed into the function, to be used by a common handler method to perform additional operations specific to that interrupt later on. For now, the reference to this function is stored in an IDT entry, that sets the base address as well as the segments selector and type:

```
idt_setup(0, (uint32_t)isr0, 0x08, 0x8E);
idt_setup(1, (uint32_t)isr1, 0x08, 0x8E);
idt_setup(2, (uint32_t)isr2, 0x08, 0x8E);
```

There are 256 possible interrupts on x86 architectures, the first 32 are system critical so consequently, need to be mapped. When all the interrupt entries for the IDT are set, the *idt_pointer* can be flushed to the IDT register using the *lidt* operational code.

Now that the CPU knows where to find the interrupt handlers, a handler for each interrupt can be created. Assembly functions can be used to push the interrupt numbers onto the stack because no interrupt data is currently given by the handler. Frustratingly, some interrupts push an error code, so there is no common stack frame; therefore, two different routines need to be made. GAS's macro facility makes writing 32 versions of interrupt routines much easier:

```
.macro isr_dummycode arg
        .global isr\arg
        .align 16
        isr\arg:
                cli
                push  $0
                push  $\arg
                jmp isr_caller_dummy
.endm
```

```
.macro isr_errorcode arg
        .global isr\arg
        .align 16
        isr\arg:
                cli
                push $\arg
                jmp isr_caller_error
.endm
```

Interrupts are disabled at the start of each macro; then after pushing the required data onto the stack, an ISR assembly handler is jumped to. This will save the processor state before calling a higher-level C handler function. After the interrupt is handled, the stack frame; holding the registers shown below, is restored.

```
struct reg_state
{       unsigned int ds;
        unsigned int edi, esi, ebp, esp, ebx, edx, ecx, eax;
        unsigned int int_num, error_code;
        unsigned int eip, cs, eflags, useresp, ss;
} __attribute__ ((aligned(4)));
```

4.2.4 Paging

On the x86, the MMU maps memory using two paging structures known as the paging directory and the paging table. Both of these tables should be 4096-byte aligned as they contain four 1024-byte entries within them. The mentioned structures are implemented in the following C structs:

```
typedef struct page_table
{
    page_t pages[ONE_K_BYTE];
} page_table_t __attribute__((aligned(FOUR_K_BYTE)));

typedef struct page_directory
{
    page_table_t *page_tables[ONE_K_BYTE];
    uint32_t physical_tables[ONE_K_BYTE];
    uint32_t physical_address;
} page_directory_t __attribute__((aligned(FOUR_K_BYTE)));
```

To enable paging on the x86 architecture, the physical address of a page directory must be loaded into the CR3 register. Before enabling, it is important to identity map physical addresses with virtual addresses so that when paging is enabled, there are pages that can be accessed without triggering a page fault. This implementation loads the CR3 register in the higher-level C language before calling an assembly function to set the PG bit in the CR0 register:

```
.global enable_paging
enable_paging:
                push %ebp
                mov %cr0, %eax
                or $0x80000000, %eax
                mov %eax, %cr0
                pop %ebp
                ret
```

The page table entries themselves; called pages are structured as specified in section 3.2.3. The pages frame attribute points to its physical address space and for this implementation, is mapped to the page using a bitmap. In comparison to a large array, bitmaps use less space and allow easy searching for free frames. There are plenty of other physical memory allocator algorithms that could be implemented, though the bitmap is effective for this level of operating system development and simple to implement.

4.2.5 The Heap

Now that paging is configured, an effective method of allocating and deallocating memory can be implemented. The challenge is tackled here by creating a few heap-based data structures to control regions of memory. The heap structure itself contains information about its own size, role, read or write access, and a table for managing its blocks and holes. The *meta_header* and *meta_footer* are used to locate the head and foot of a block or hole; depending on the bit set for the header structs free attribute:

```
typedef struct
{
    uint32_t magic;
    uint32_t size;
    uint8_t  free;
} meta_header_t;

typedef struct
{
    uint32_t magic;
    meta_header_t *head;
} meta_footer_t;
```

```
typedef struct
{
    meta_table_t table;
    uint32_t start_address;
    uint32_t end_address;
    uint32_t max_address;
    uint8_t role;
    uint8_t readable;
} heap_t;
```

The heap table itself is ordered by comparing the values of the hole or block headers size attribute and arranging them in ascending order. It is insertion sorted and remains in a sorted state between calls; the *unknown_t* type is used to store anything that can be cast to *void**; like a *uint32_t* or any pointer. The ordering behaviour is encapsulated within the *order_t* function pointer to compare the size of holes or blocks:

```
typedef void* unknown_t;
typedef char (*order_t)(unknown_t, unknown_t);

typedef struct
{
    unknown_t *pointer;
    order_t order;
    uint32_t max_size;
    uint32_t size;
} meta_table_t;
```

Because we are using paging, to allocate space for the heap, pages must be allocated before initialisation. Once the pages have been mapped to the physical heap address frames, the interrupt handler for page faults can be registered and the page directory loaded. After all this, the heap can finally be initialised with paging enabled, and no page faults exceptions should be thrown.

Before the heap is initialised, memory can still be allocated using memory allocation by placement address. This is optimal for time and space allocations but does not combat the issue of deallocation. This implementation uses two variations of the well known C library *malloc()* function: *malloc_virt()*; to allocate virtual memory, and *malloc_phys()*; to allocate physical memory. In *malloc_virt()* the size requested is simply allocated in the kernel heap and then an address pointer is returned. The *malloc_phys()* function, on the other hand, will assign a page to a page frame using a parsed physical address parameter, before returning the address pointer.

```
uint32_t malloc_virt(size_t size, uint8_t align)
{
  if (kernel_heap != 0)
  {
      // TODO: allocate stuff here once heap is initialised
      void *address = allocate(size, (uint8_t)align, kernel_heap);
      return (uint32_t)address;
  }
  else
  {
    if( align == 1 && (break_point_address & ALIGNMENT))
        page_align();

    uint32_t mem = break_point_address;
    break_point_address += size;
    return mem;
  }
  return 0;
}
```

The *allocate()* function for the heap works by firstly, searching the heap table to find the smallest hole that can fit the size requested. If there is no hole large enough, then the heap is instructed to expand; bounded by a max size, before recursing and trying to find a hole again. If the smallest hole has a lot of free space, then it is split in half, and a new hole is written to the heap table. At the end of all this resizing and hole formatting, the address plus the size of the *meta_header* is returned to the user.

The *deallocate()* function, is a little more complicated. It involves a unifying algorithm that takes two unallocated adjacent holes and amalgamates them back into one larger hole. When deallocating a block, look at what is left of the block, if it's a footer, then follow the footer pointer to the header. If the headers free attribute is not set then the size attribute can be modified by adding the block being merged. The headers pointer can then be changed to point to the previous blocks footer instead. Unifying from the right is slightly different but follows the same principle.

4.2.6 Multitasking

Lastly, one of the final hurdles of creating an operating system is providing it with the ability to run multiple tasks seemingly at once. This begins by defining a C task struct to hold all the attributes of a process. The *reg_state *r* pointer should contain the saved state of the processor registers before a task interrupt; because paging is enabled, the physical address of whatever directory the task is in should be saved in '*cr3*'. To implement the round-robin scheduling policy later, the task must contain a pointer to the next task. The function pointer is simply to pass an operation, which in this case, is to display the stored task message.

```
typedef struct task {
  uint8_t id;
  void (*function)(void *);
  void *message;
  uint32_t *stack;
  struct reg_state *r;
  struct task *next;
  uint32_t cr3;
} task_t;
```

Conveniently, there already exists a mechanism to retrieve the register state when a task needs to switch. The common C interrupt handler created in section 4.2.3, already pushes the required contents onto the stack. Therefore, when an IRQ0 is triggered by the hardware timer, the task state can be saved and switched. With a frequency of 100hz set, an interrupt will occur every 0.01 second, so when the remainder of the number of ticks divided by the set timeslice constant is equal to 0, a task switch is triggered. This means that if the timeslice is set to 100, a task switch will be triggered every second.

```
static void timer(struct reg_state r)
{
  ticks++;
  if(ticks % TIME_SLICE == 0){
    if(scheduling == 1)
      switch_tasks(r);
    seconds++;
  }
}
```

Now that task switching is activated, the *switch_tasks()* function can take the saved register structure as an argument when called by the IRQ0 handler. It operates by firstly, copying the saved register state from the interrupt event to the task structs *reg_state *r*, using the *copy_memory()* function. It will then check if the current task is set and if so, sets it to execute and sets the current task equal to the next queued task, for when the function recurs. After, it copies the current processes register state back to the stack and switches the page directory to the directory stored in the current task, demonstrated by the following:

```
void switch_tasks(struct reg_state r)
{
    /* copy the saved register from the interrupt timer
       into the task struct */

    copy_memory(current_task->r, r, sizeof(reg_t));

    if(current_task->next != 0){
      execute_task(current_task);
      current_task = current_task->next;
    }

    /* copy the current processes register state back
       to the stack */

    copy_memory(r, &current_task->r, sizeof(reg_t));

    switch_directory(current_task->cr3);
}
```

Chapter 5

Testing

Now that all the features of a minimal x86 operating system have been implemented and a bootable image has been built, testing can commence. All testing in this paper is done using QEMU, although there are other emulating platforms that can be used, such as BOCHS. Each emulator has its own unique simulations and flexible configurations that can be taken advantage of at different stages of development. However, QEMU allows quick emulation and is suitable for testing a minimal operating system. The following sub-chapters have been selected for testing because they are milestones in operating system development; together, they supply the necessary attributes of a minimal system.

5.1 Interrupt Handling

To validate that the minimal operating system can correctly handle an interrupt, a software thrown interrupt can be used for testing. By entering *'__asm__ volatile("int $0x08)'* in the *kernel.c* file after interrupts have been initialised; a double fault exception will be triggered. As you can see in figure 17, the fault is handled, and a double fault exception function that is registered with interrupt eight is called. This displays useful register information that can be used for debugging.

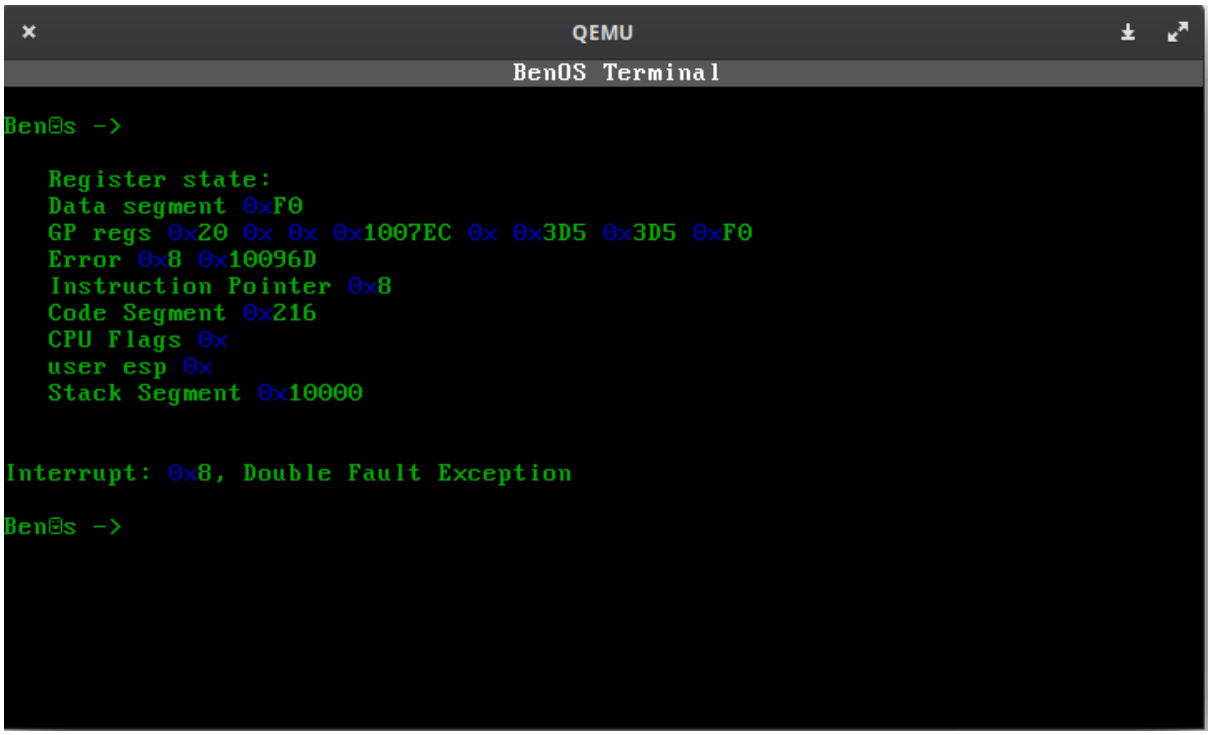


Figure 17 – A screenshot of BenOS handling a double fault exception.

5.1.2 Paging

To test paging, a variable with a pointer can be assigned to an illegal address space; such as *0xAAAAAAAA*, that has not been assigned a page. When the variable pointing to that address space is used, it will cause a page fault exception which is caught and handled in the *paging.c* code. Useful information about why the fault occurred is then outputted to the terminal and system execution is stopped to prevent any further damage.

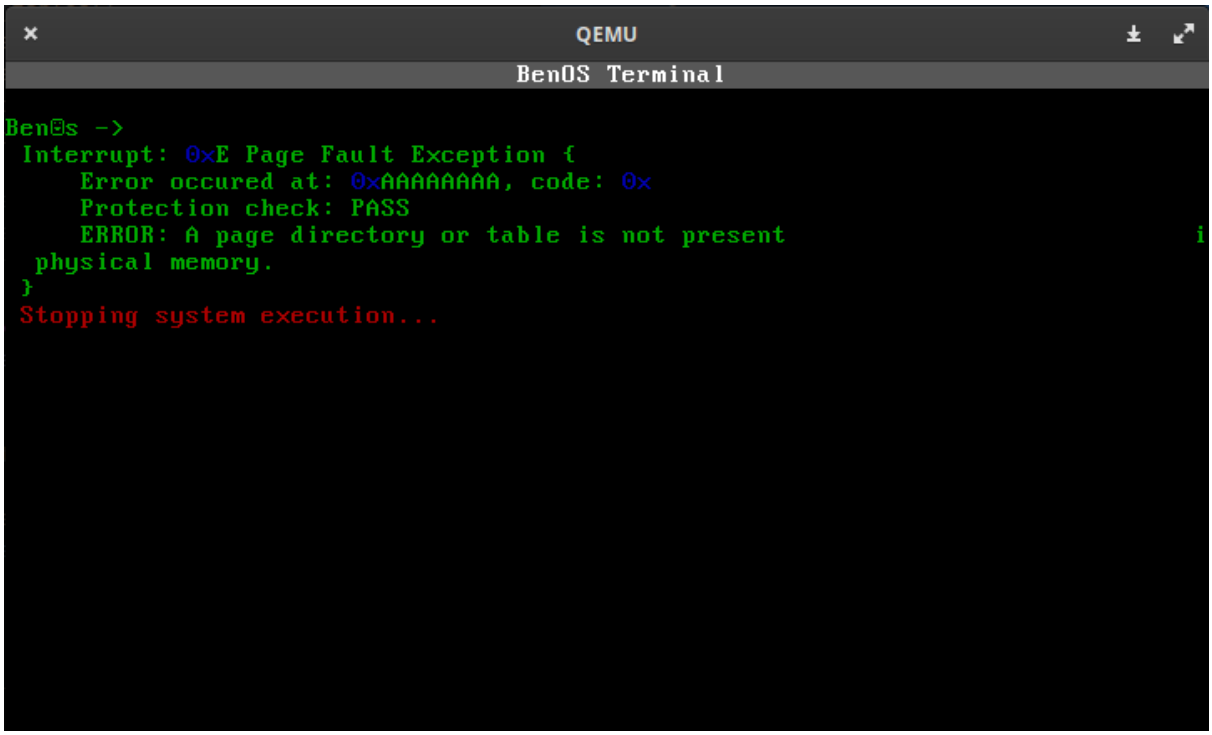


Figure 18 – A screenshot of BenOS handling a page fault exception

5.1.3 Memory Allocation

To evidence the minimal operating systems ability to allocate and deallocate memory, a few lines of code had to be produced; shown in figure 19. The code assigns various sized chunks of memory to variables: *a*, *b*, *c*, and *d*. After allocating memory to the first three variables, their starting addresses are outputted to the terminal. Two of those variables are then released from memory using the *deallocate()* function. The final variable is assigned and outputted to the terminal for assessment. In figure 20, the terminal shows the start address of each allocated chunk. The address of variable *d* confirms that the variables b and c where deallocated because the

```
uint32_t a = malloc_virt(16,0);
uint32_t b = malloc_virt(8,0);
uint32_t c = malloc_virt(64,0);

println(" a %h", a);
println(" b %h", b);
println(" c %h", c);

deallocate(c);
deallocate(b);
uint32_t d = malloc_virt(12,0);
println(" d %h", d);
```

Figure 19 – A screenshot of the code used to allocate and deallocate memory.

starting address is *0xB0080030* which is 16 bytes from the starting stack address of *0xB008000C*.

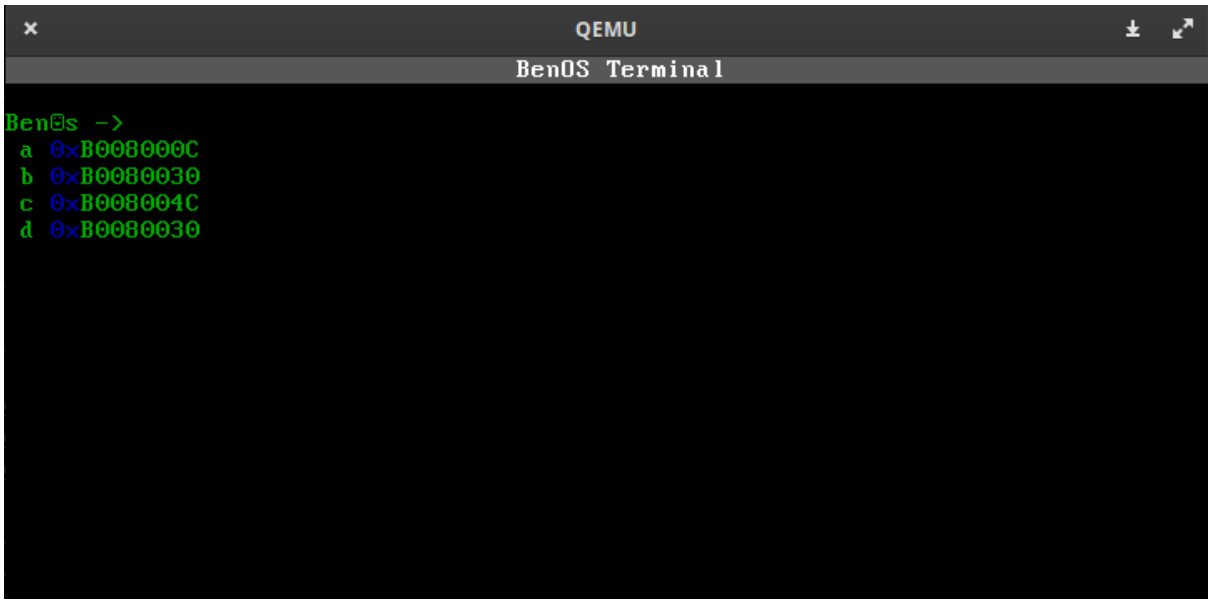


Figure 20 – A screenshot of BenOS dynamically allocating and deallocating memory from the heap.

5.1.4 Scheduling

As a demonstration of the minimal operating systems ability to switch between processes, a large time slice; two seconds, was set in order to visualise the process switching clearly. The process id and starting address of the tasks dedicated address spaces are outputted to the terminal to for context. Each task is allocated a *0x1000* sized chunk of address space and is placed in a dynamic round-robin queue for execution.

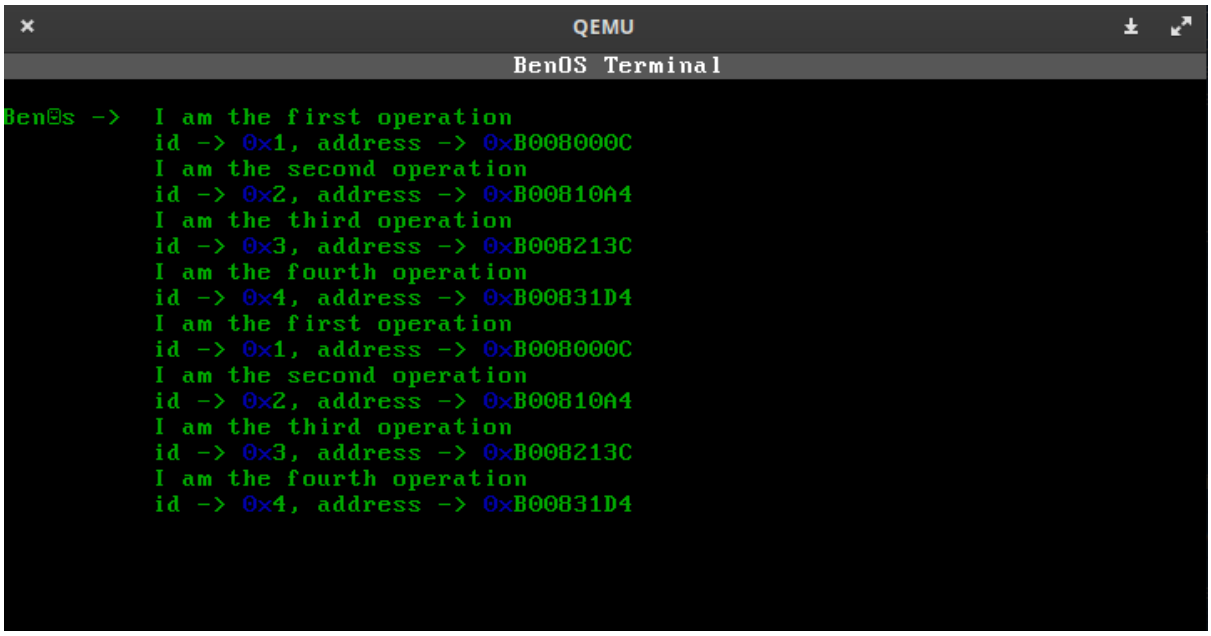


Figure 21 – A screenshot of BenOS switching between processes using the round robin policy.

Chapter 6

Conclusion

6.1 Design vs Implementation

In brief, the design was almost entirely followed by the implementation, though a few decision were changed for practicality. For instance, the memory management techniques discussed in section 3.2.3 were implemented as planned, whereas the multitasking implementation did not cover every detail specified by the section 3.4 design. The implementation could have included process states and threads as mentioned, though due to a shortage of time, said inclusions could not be made functional. In the end, the design was mostly followed, though a few minor components had to be missed in order to finalise the project.

6.2 Future Improvements

As operating systems are substantial in both size and versatility, there is ample opportunity for improvement and further development. One of the main features that this implementation misses out on is a virtual file system. Support could have been added for a filesystem; such as an ext2 or tar, to manage secondary storage, though lack of time prevented this. It would have been practical in helping demonstrate the capabilities of multitasking by having processes that read and display files within the filesystem. If development were to continue, a file system would certainly be implemented, as an operating system should be capable of permanently storing data.

To follow convention and allow the operating system to be compatible with existing software, a few function names could be altered to match the C standard library. For example, the *set_memory()* and *copy_memory()* functions can be changed to the C standard *memset()* and *memcpy().* Coding standards reduce program complexity and thereby reduce errors, allowing better maintainability and improved consistency. Additionally, the operating system offers compatibility for a user-level through privilege settings in memory management and architecture, yet no actual user space exists. A multi-user environment should be supplied to improve system security and to provide a safer working environment with personal secondary storage.

To summarise, there are endless possibilities for improvement when developing an operating system. A few areas for growth and development have been mentioned above, though a multitude of additional topics could have been covered: support for other architectures, demand paging, enhanced graphics, networking, a user login screen and audio, are just a few examples of areas for improvement or expansion. So, whether it be to better performance or to improve functionality, there is always room for modification and further development within an operating system.

6.3 Review of Project Aims

- **Provide a synopsis of the attributes that form an operating system.**

The background section of this paper provides detailed documentation about operating system core features. The paper clearly explains these concepts and hopefully supplies the reader with a strong foundation of knowledge that they can use for the preceding chapters. Unfortunately, operating systems are substantial, and therefore, all possible attributes could not be covered considering the time constraints. Research into networking, audio and video are a few examples of concepts that could have been covered if more time was available. So, in summary, the objective was fulfilled by supplying an overview of a minimal system's attributes, though more coverage could have been provided.

- **Investigate the necessary requirements needed for a basic operating system to function for a specific architecture.**

Chapter 2 of this paper discusses the advantages of developing with the x86 architecture. It covers various protection and supporting features provided by the architecture and discusses why it would be ideal to focus development towards it. Although it is made clear from the beg that the system would be developed for the x86, it would have been informative to analyse other existing architectures also. Moreover, the kernel structure set up in Chapter 3 provides room for modification, which allows support for other architectures and provides the means for further expansion and support. Overall, the primary objective was satisfied, and the reader is provided with requirements to deliver a minimal operating system for a specific architecture.

- **Review the previously researched requirements and formulate a rough design with justified decisions.**

After analysing the requirements for a minimal operating system, Chapter 3 provides instructions on how to formulate a plan for development. The approaches given to tackling certain design decisions have been justified and explained. It is intended only to be a rough guide that aims to assist by providing justified approaches to various operating system concepts. Occasionally, the design and researched requirements do seem to merge, though this is because, at times, a minor amount of additional background information is required to explain and justify why the design is the way it is. To conclude this aim, the groundwork for developing an operating system is provided, with reasons justified, although at times the difference between the design and background sections are not completely clear-cut.

- **Produce a procedural guide and implementation of a minimal operating system, including dynamic memory allocation and effective multitasking.**

In Chapter 4, a step-by-step guide of how to implement a minimal operating system is supplied, giving a detailed review of how to develop key operating system features with

examples, chronologically. Most importantly, a great depth of coverage on dynamic memory management and effective task scheduling is provided; the core hurdles in developing an effective operating system. Regrettably, no time was available to cover one of the very first steps after building the environment, the graphical display. An additional section could have been added that provides example code of how to display graphics on-screen. To summarise, a detailed procedural guide is provided with the mentioned core features, though a few intermediary steps in-between would help the reader transition between stages of development.

- **Examine whether the implemented operating system conforms to the proposed design and discuss how it could be improved in the future.**

The comparison of design and implementation is covered in section 6.1.1, of which addresses a few minor issues with the design of multitasking and the manner in which it was implemented. After assessing the two chapters, the preceding section 6.1.2, focuses on what improvements could be made and how the operating system can be developed further. Due to operating system development being such a vast topic, there exist many ways that development could be extended. Therefore, only a few examples are explained in detail and others listed, as it is a large topic to cover.

6.4 Final Comments

In summary, with all the primary aims of this paper satisfied and a functioning minimal operating system built, the project can be successfully concluded. Though it has been challenging and frustrating at times, I am grateful for the opportunity to tackle it. It has left me with a feeling of self-accomplishment and a much greater depth of understanding. Moreover, my programming skills have been greatly enhanced, and my understanding of C as a powerful and manipulative language has advanced. I will continue the development of BenOS in my own time for self-learning and in the hope that it will progress into something greater than it currently is. I hope that one day this project can be used by others to assist and enlighten them about operating system concepts and development.

Bibliography

[1] A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts 9th Edition, New Jersey: John Wiley & Sons, 2014, pp. 345 - 800, ISBN: 978-1-118-09375.

[2] M. Kanellos, “Moore's Law to roll on for another decade,” CNET, 11 February 2003. [Online]. Available: https://www.cnet.com/news/moores-law-to-roll-on-for-another-decade/. [Accessed 29 February 2020].

[3] A. S. Tanenbaum, Modern Operating Systems 3rd Edition, New Jersey: Prentice Hall, 2009, pp. 12 - 357, ISBN: 978-0-13-813459-4.

[4] J. Catsoulis, Designing Embedded Hardware, Brisbane: O'Reilly Media, 2005, ISBN: 0-596-00755-8.

[5] L. B. Das, The X86 Microprocessors: Architecture and Programming (8086 to Pentium), New Delhi: Dorling Kindersley, 2010, p. 561, ISBN: 978-93-325-3682-1.

[6] J. G. Spooner, “Intel: A quarter-century of x86,” CNET, 10 June 2003. [Online]. Available: https://www.cnet.com/news/intel-a-quarter-century-of-x86/. [Accessed 29 February 2020].

[7] Intel Corporation, “Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A,” 2016. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html. [Accessed 20 January 2020].

[8] M. Matz, J. Hubicka, A. Jaeger and M. Mitchell, “System V Application Binary Interface AMD64 Architecture Processor Supplement,” 17 November 2014. [Online]. Available: https://uclibc.org/docs/psABI-x86_64.pdf. [Accessed 4 March 2020].

[9] Unified EFI, “Unified Extensible Firmware Interface Specification,” January 2016. [Online]. Available: https://uefi.org/sites/default/files/resources/UEFI%20Spec%202_6.pdf. [Accessed 5 March 2020].

[10] D. Elsner and J. Fenlason, “Using as The GNU Assembler,” 1991. [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html. [Accessed 1 March 2020].

[11] T. Rothwell and J. Youngman, “The GNU C Reference Manual,” 2007. [Online]. Available: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf. [Accessed 1 March 2020].

[12] S. Chamberlain and I. L. Taylor, “The GNU linker,” 1991. [Online]. Available: https://www.eecs.umich.edu/courses/eecs373/readings/Linker.pdf. [Accessed 1 March 2020].

[13] G. Matzigkeit, Y. K. Okuji, C. Watson and C. D. Bennett, “the GNU GRUB manual,” 24 June 2019. [Online]. Available: https://www.gnu.org/software/grub/manual/grub/grub.pdf. [Accessed 1 March 2020].

[14] R. Love, Linux Kernel Development: A thorough guide to the design and implementation of the Linux Kernel (Developer's Library) 3rd Edition, Crawfordsville, Indiana: Pearson Education Inc, 2010, Ch. 1 , ISBN: 978-0-672-32946-3.

[15] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram and U. Drepper, “The GNU C Library Reference Manual,” 1993. [Online]. Available: https://www.gnu.org/software/libc/manual/pdf/libc.pdf. [Accessed 27 February 2020].

[16] AMD64 Technology, “AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions,” September 2019. [Online]. Available: https://www.amd.com/system/files/TechDocs/24594.pdf. [Accessed 26 February 2020].

[17] D. Lea, “A Memory Allocator,” 4 April 2000. [Online]. Available: gee.cs.oswego.edu/dl/html/malloc.html. [Accessed 10 March 2020].

[18] R. M. Stallman, R. McGrath and P. D. Smith, “GNU Make: A program for Directing Recompilation,” January 2020. [Online]. Available: https://www.gnu.org/software/make/manual/make.pdf. [Accessed 26 January 2020].

[19] N. Ishkov, “A complete guide to Linux process scheduling,” February 2015. [Online]. Available: https://trepo.tuni.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf. [Accessed 27 February 2020].

[20] P. A. Janson, Operating Systems Structures and Mechanisms, London: Academic Press Inc, 1985, p. 3, ISBN: 0-12-380230-x.

Feedback comments

FIRST MARKER : Comments on written report:b'The report has a good structure and is generally well written, although there is some odd wording in places.\r\rThe report has a good degree of technical depth but is too focused on guiding the reader through the development and seems to miss context and lacks a proper narrative on why particular decisions were made. The implementation section, for example, seems more a discussion on the build process than on realizing a design. The split between design and implementation should also be more distinct.\r\rThe discussion of testing is very brief, which is odd considering the considerable testing that must have been done throughout the project.\r\rThere is a lot in the report, but it seems rushed and should be refocused.' --- Viva Comments:b'This is a complex project that involved navigating some significant pitfalls and it is clear that a lot is working. It is a significant achievement getting so far -- it is a shame that time ran out when it did as there is not much more to do before this would be a usable system.\r\rThe viva included a presentation and demonstration run over MS Teams. The presentation covered the background to the project, the major elements within the system, and ideas for future work. Ben came across well and made some good points on both design and implementation.\r\rThe demonstration included booting a PC with BenOS, VGA output, keyboard input, interrupts, task scheduling, paging, and heap allocation. It was interesting to see Ben had moved the entire build chain from Linux to MS Windows to make it easier to demonstrate on Teams.' SECOND MARKER: Comments on written report:b'A comprehensive report, documenting the principles upon which operating systems are founded, and how the student applied them to create a minimalist operating implementation for the x86 architecture. \r\rThe student documents the implementation of a keyboard driver, VGA (text mode) graphics driver, memory management algorithms (MMU protected virtual heap and stack), and preemptive scheduler. The student also documents plans for a thread scheduler and file system implementation.\r\rThe report is clear and concise, and records the results of a highly challenging piece of work. However, it is rather implementation focussed and would have benefited from deeper reasoning around the design alternatives that could have been chosen, and a more thorough evaluation.\r' --- Viva Comments:b'An excellent viva, given via a Teams meeting using screen share. The student gave a concise and focused presentation that clearly outlined the work undertaken. This was followed by a demonstration of the scheduler and memory allocator developed. This was illustrated through practical test cases running on a Qemu x86 emulation.\r\rThe student provided a clear and objective demonstration and responded well to all questions posed.'