Incitatus Operating System
Instructional OS for CS Students

Ali Ersenal - 092648605
Dr. Nick Cook
Newcastle University

May 5, 2013

A dissertation submitted for the degree of

Computing Science BSc

Word Count: 13308

I declare that this dissertation represents my own work except where otherwise stated.

Acknowledgements

I would like to thank my loving parents for their never-ending support and my supervisor, Dr. Nick Cook, for his guidance. I would also like to express my gratitude to the OSDev community for helping me overcome many obstacles throughout the project.

Abstract

An operating system is a complex software system which manages the underlying hardware resources and provides services to upper layer applications or programs. Due to their nature, inner workings of an operating system tend to be confusing and complicated.

The goal of this dissertation is to produce an operating system in order to introduce operating system concepts to computer science students; augmenting their understanding through hands-on experience. Implementation details of the produced operating system are presented along with discussions on how it differs from mainstream operating systems.

The result of this project is an operating system which is functional, concise, modular and extensible; allowing students to learn, explore and extend operating system primitives.

Contents

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 Genesis

Computer science students are nowadays taught at a higher level of abstraction than they were back in the past. The advent of higher level languages, frameworks and safer virtual machines brought much needed ease and comfort to computer science students; instead of fiddling with low-level implementation details, they are able to focus on the real meat which is the actual computer science concepts.

While this is not necessarily a wrong approach, it brings some issues with it. Through my own observations as a fellow student, I feel that computer science students are becoming more and more detached from what goes behind the scenes; unable to complete the whole picture. Being aware of the whole *stack* (from kernel to user written applications) not only grants students the knowledge to fit in the missing pieces, but also helps them produce higher quality code.

The idea for this project first began out of pure curiosity, as I started wondering how a simple Java call such as *System.out.print()* really works. Soon enough, I found out that the seemingly trivial call lent itself to a surprisingly complex structure; moving from Java calls to Java Native Interface calls to operating system calls.

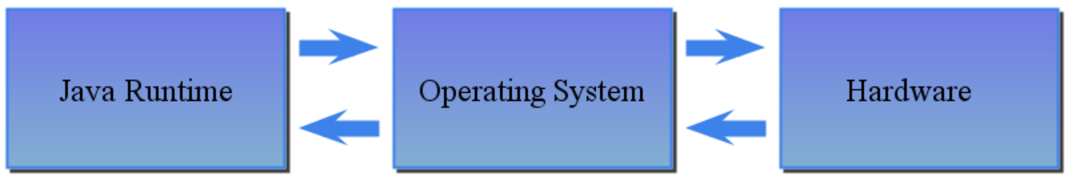


Figure 1.1: Java call stack

After first handedly (as a computer science student) experiencing a lack of confidence over the internal workings of a simple Java call, I set out on a quest to understand the *stack*. Operating systems are a major part of the *stack*, therefore I turned my focus to understand how they are implemented. For this purpose, I chose to implement a functional (albeit minimal) operating system from scratch. Hence, this dissertation can also be interpreted as a diary of a Java-era computer science student delving into operating systems development.

1.2 Aims and Objectives

The main aim of this dissertation is to present how modern operating systems work internally and to ease the introduction of operating system concepts (therefore the *stack*) to computer science students by implementing an extensible operating system. In order to achieve this aim, the project itself must be able to satisfy the following objectives:

- Supplement operating systems classes, build up on theory by providing hands-on experience on common OS primitives.
- Demonstrate how modern monolithic operating systems on a modern micro-architecture operate.
- Evaluate the developed operating system for its usefulness in teaching.

1.3 Requirements

To satisfactorily meet the objectives outlined above, the following system requirements are necessary:

1. The operating system must be concise and to-the-point, main focus is to enhance comprehension of OS concepts. Therefore performance or hardware compatibility must not be the primary focus of the project, which need lots of development time and result in a complex/bigger code base.
2. The code base should be as small as possible and at the same time as easy to understand. Students should not feel lost while navigating the code base. For this reason coding style must be as clear and tidy as possible.
3. The operating system must be extensible. Students should be able to extend or alter the functionality, encouraging them to experiment and explore.

4. A modular approach makes it easier for students to figure out a certain aspect of the operating system, therefore the operating system must be modular and should provide dependency control between modules. Modules represent the OS primitives such as process scheduler, drivers, memory managers. A student should be able to implement and plug in his own modules in place of the default ones.

1.4 Project Management and Plan

Given below is the proposed project plan. Tasks concerning the development of the operating system have been met on time and as planned. Majority of the work in the first semester involved kernel space development. Whereas the user space development took place in the second semester.

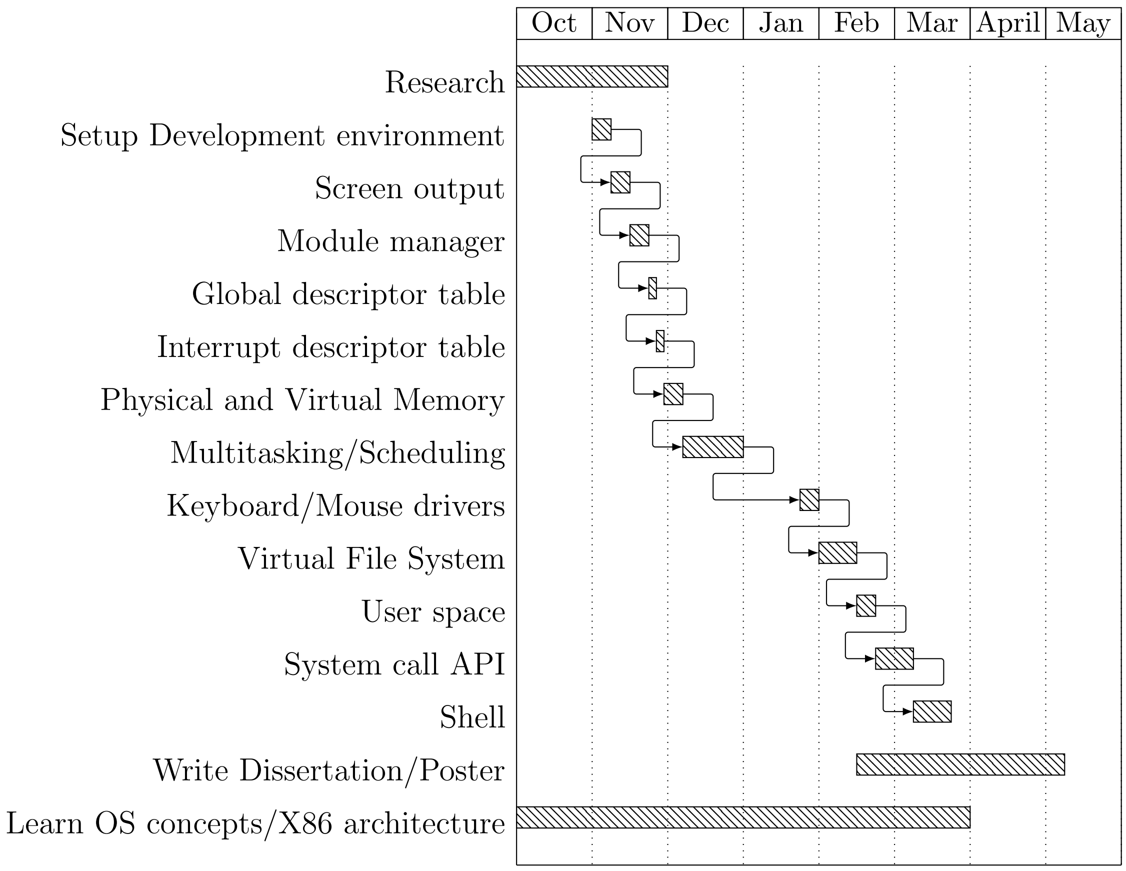


Figure 1.2: Proposed project time plan

No specific software development model was used while developing the operating system. Yet, care was taken to keep the development organized by utilising the git[5] version control system and a to-do list.

1.5 Document Map

Chapter Two
Presents a detailed review of the background material relevant to this project.

Chapter Three
Provides a description of the system architecture.

Chapter Four
Outlines how the proposed design is implemented.

Chapter Five
Provides instructions on how to modify and build Incitatus.

Chapter Six
Presents a summary of the results produced by the project and evaluates the proposed solution against the original objectives.

Chapter Seven
Provides a conclusion by analysing the whole project and discussing how well the final solution fulfils the objectives.

Chapter 2 - Background Research

This chapter contains a summary of the background research carried out during the project. The chapter starts out by briefly explaining the relevant operating system concepts, continues by introducing and critically evaluating the existing work and concludes with a brief discussion on tools/technologies that have been used during the development of the operating system. Operating systems development is an extremely vast topic, therefore only the most important and relevant parts of the research have been included.

2.1 Operating System

An operating system is a complex software structure consisting of interconnected components which manage the underlying hardware resources and provide services to upper layer applications (or programs). Nowadays, many electronic devices (smart phones, computers, game consoles, robots, satellites, etc.) contain an operating system[25]. Simply put, an operating system is just another layer of abstraction on top of other layers (circuits, gates, micro-architecture, etc.).

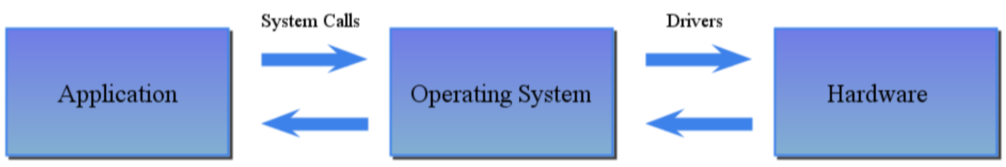


Figure 2.1: A typical operating system placement

In Tanenbaum's words:

> *"It is hard to pin down what an operating system is other than saying it is the software that runs in kernel mode and even that is not always true. Part of the problem is that operating systems perform two basically unrelated functions: providing application programmers*

(and application programs, naturally) a clean abstract set of resources instead of the messy hardware ones and managing these hardware resources."[22, p. 3-4]

2.1.1 Kernel Design

Kernels make up the main part of an operating system. Kernels, typically, can be categorised according to their design and general architecture. This subsection presents the two prominent kernel designs: monolithic and micro kernel.

Monolithic Kernel

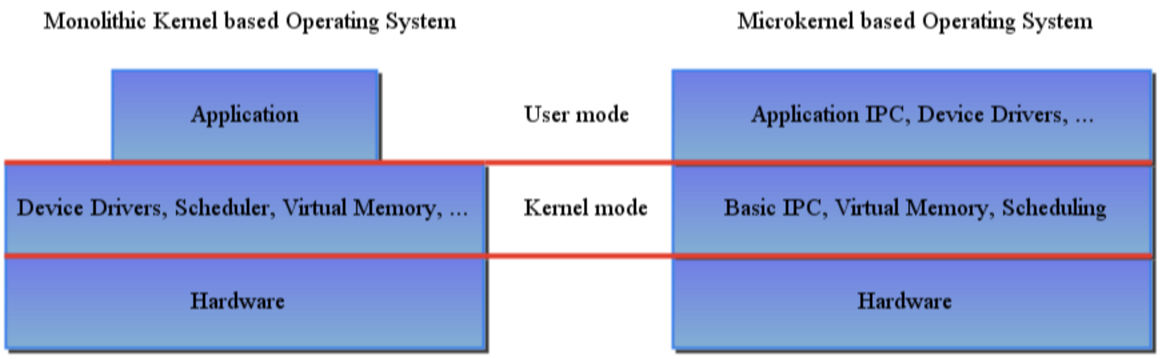
Monolithic kernel is the traditional kernel design. Operating system services are executed in the kernel mode and system calls provide an interface to the user applications (which are run in user mode). Monolithic kernels tend to be faster than microkernels as the runtime code overhead is higher in microkernels. According to Ken Thompson, who designed and implemented the original Unix operating system, it is relatively simpler and easier to implement than microkernels[23]. The most critical drawback of monolithic kernels is reliability. Because of the fact that all the operating system services are executed in kernel mode, a bug in a device driver has the potential to bring down the whole system.

A typical system call (such as opening a file) in a monolithic operating system behaves as follows:

1. User application pushes the system call parameters.
2. User application raises a software interrupt, switching to kernel mode.
3. System call interrupt handler looks up the requested service and executes it with the given parameters.
4. After that, the execution returns back to user mode; continuing where it was left off.

Microkernel

Microkernel aims to improve the reliability of the operating system by keeping most of the operating system services in user mode as servers (such as device drivers). Only the most fundamental services are run in kernel mode (such as context switching, low level memory management). Servers communicate with each other by sending messages. Unlike monolithic kernels, a bug in a device driver (a server in microkernel) does not crash the whole system. Servers or, less frequently, user applications still make use of a system call interface (albeit thinner than a monolithic kernel where all the operating system services are included) for tasks such as requesting more memory or basic inter process communication (IPC).



Monolithic Kernel based Operating System
Microkernel based Operating System
Application
User mode
Application IPC, Device Drivers, ...
Device Drivers, Scheduler, Virtual Memory, ...
Kernel mode
Basic IPC, Virtual Memory, Scheduling
Hardware
Hardware

Figure 2.2: Monolithic kernel versus Microkernel

Andrew S. Tanenbaum, a proponent of microkernels, describe microkernels as:

> *"The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules, only one of which the microkernel runs in kernel mode and the rest run as relatively powerless ordinary user processes. In particular, by running each device driver and file system as a separate user process, a bug in one of these can crash that component, but cannot crash the entire system. Thus a bug in the audio driver will cause the sound to be garbled or stop, but will not crash the computer. In contrast, in a monolithic system with all the drivers in the kernel, a buggy audio driver can easily reference an invalid memory address and bring the system to a grinding halt instantly."*[22, p. 63]

2.1.2 Process Management

In modern multitasking operating systems, process management takes up a very critical role. Process management allows an operating system to:

- Create or destroy processes
- Manage allocation of resources to processes
- Oversee inter process communication

- Decide the workload of processes through scheduling

Process Management Model

Process management models allow the operating system to "know" the state of a process. This knowledge is critical as it guides the scheduler. Popular process management models include:

- Two-state process management model
- Three-state process management model
- Five-state process management model

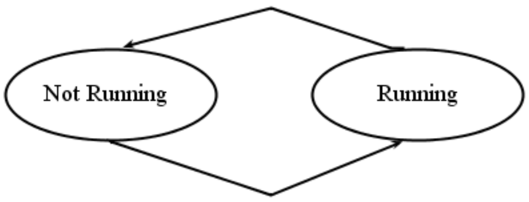


Figure 2.3: Two-state process model

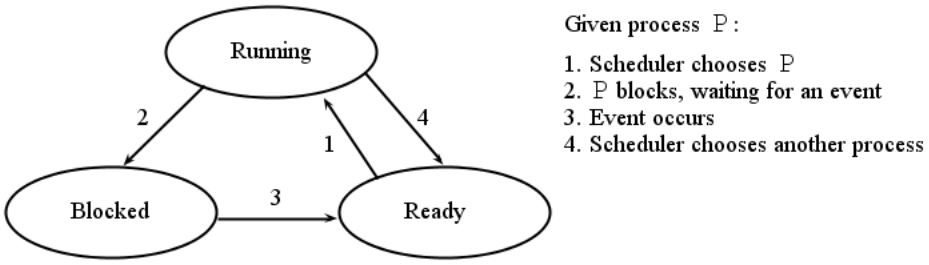


Figure 2.4: Three-state process model

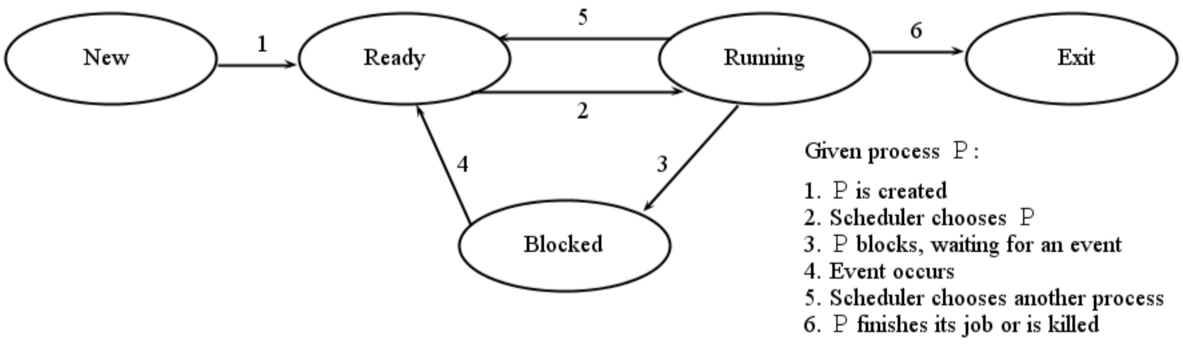

New
Ready
Running
Exit
Blocked
1
2
3
4
5
6
Given process P:
1. P is created
2. Scheduler chooses P
3. P blocks, waiting for an event
4. Event occurs
5. Scheduler chooses another process
6. P finishes its job or is killed

Figure 2.5: Five-state process model

First Come First Served Scheduling

FCFS is the simplest process scheduling algorithm. Processes are assigned for execution in the order they request it. Operating systems with this type of scheduling are nonpreemptive; executing processes one after the other. This means that the average waiting time for an execution of a process is usually quite long.

Round-Robin Scheduling

Round robin scheduling is one of the simplest preemptive process scheduling algorithms. Each process is assigned an equal time slice (*quantum*) and allowed to execute only within this time slice. If the process does not finish (or finishes) its job before its time slice ends, the scheduler switches to the next process. Thus, round-robin scheduling ensures a fair distribution of CPU time across processes.

2.1.3 Memory Management

Even the simplest operating systems require a way of managing the computer memory. Memory management in a typical operating system tends to be structurally segmented. At the lowest level (often called the physical memory manager), the memory manager works with the physical memory which is the RAM. The next layer, using the services provided by the physical memory manager, takes care of managing the virtual memory; handling the address spaces. Virtual memory acts as a layer of abstraction, giving the illusion of contiguous memory. At the top of the pyramid lies the *heap*, a large pool of memory managed by heap memory managers. Heap memory managers provide a high level access to dynamic allocation and deallocation of memory. Often, in operating systems, kernel side and user side have two completely different heap managers. This is because they both have different allocation or deallocation patterns.

2.2 Existing Work

This section introduces some of the existing work relevant to this project and critically evaluates them on their effectiveness at being an instructional operating system. Table 2.1 compares the main features of such existing work against the proposed operating system, *Incitatus*.

Operating System	CPU Architecture	Kernel Architecture
Minix 2.0	x86, SPARC	Microkernel
Nachos	MIPS	Microkernel
Pintos	x86	Microkernel
Incitatus	x86	Monolithic kernel

Table 2.1: Instructional operating systems comparison

2.2.1 Minix

Minix (mini-unix)[21] is an operating system developed by professor Andrew S. Tanenbaum for educational use. The first version of Minix was released in 1987 along with its source code[24]. As well as being instructional, Minix aims to be a reliable operating system, thus features a microkernel architecture as opposed to a monolithic one. Minix version three was released in 2005 as a highly reliable and modern substitute to other mainstream operating systems.

Being a microkernel, Minix contains only the most essential operating system functionality in the kernel space such as the system timer driver. All other device drivers are located in the user space and run as separate user processes. Thus, a faulting driver does not crash the whole system. User space drivers provide services for an upper layer, which is also located in user space. This layer contains *servers* which do most of the work of the operating system. Servers manage file systems, processes, network, etc... If somehow a server becomes unable to operate, it is *reincarnated* by the reincarnation server. Thus, the system is said to be *self healing*[22, p. 63-64].

As opposed to version three, Minix versions one and two were created with students in mind; therefore version three has a relatively larger code base and complexity. Given that versions one and two were released in 1987 and 1997[24], they may be considered quite old.

2.2.2 Nachos

Nachos[11] operating system was designed by Thomas Anderson for teaching in operating systems courses. It was written in the C++ programming language for the MIPS architecture.

Nachos is implemented in a subset of C++. Object-oriented programming is becoming more popular, and we found that it was a natural idiom for stressing the importance of modularity and clean interfaces in building operating systems.

To simplify matters, we omitted certain aspects of the C++ language: derived classes, operator and function overloading, and C++ streams. We also kept inlines to a minimum. Although our students did not know C++ before taking our course, we found that they learned the language very easily."[11, p. 3-4]

Compared to many modern operating systems, Nachos is a simulated operating system as it runs on top of a host operating system and an MIPS simulator. A simulated environment is not necessarily detrimental to learning operating system concepts but it may discourage the students as it cannot run on real hardware.

2.2.3 Pintos

Pintos[19] is an instructional operating system for undergraduate computer science students. It was developed for the recent X86 architecture and is meant to replace Nachos. Pintos' primary aim is to implement OS primitives in a simple way so that students may easily study operating system concepts.

"Although Pintos follows in the tradition of instructional operating systems such as TOY OS [9], Nachos [6], OS/161 [12], and GeekOS [13], PortOS [2], BLITZ [16], JOS [1], or Yalnix [1], we believe that it is unique in two aspects. First, Pintos runs on both real hardware and in emulated and simulated environments.1 We believe that having the ability to see the outcome of their project interact with actual hardware increases student engagement. Second, we have created a set of analysis tools for the emulated environment that allows students to detect programming mistakes such as race conditions."[19, p. 1-2]

2.3 Tools and Technologies

While carrying out the background research, a range of tools and technologies that are suitable for use in the project were identified. This section acts as an introduction to such tools or technologies and justifies why they were chosen in the first place.

Programming Language

Typically, operating systems development requires the use of low-level languages as direct memory manipulation and no runtime is a necessity. Most programming languages do not allow direct modification of registers which is required for this project. Thus, assembly language[13] is a clear choice as it is:

- Low level, able to manipulate memory and registers
- Runtime-less
- Efficient

In spite of its merits, assembly language is hard to maintain and debug. Thus, it is put in action only when it's absolutely necessary. C programming language[20] is used as a higher level replacement for the assembly language as it strikes a good balance between maintainability and low level functionality.

GCC[3], Nasm[7], GNU Binutils[1] were used in the project to compile and link the code to 32-bit x86 architecture.

Emulator

It is very critical to debug the operating system while developing it. Given that incremental testing on physical hardware is extremely cumbersome, emulators provide a practical solution to this problem by being able to emulate the operating system on a host operating system. After a brief research, QEMU[8] and Bochs[2] were selected as suitable emulators as they both support emulation for the x86 architecture.

Debugger

Debugging is a crucial element especially in low-level development as it is quite easy to unintentionally produce a critical bug. GDB[4] was used throughout the project to find and fix those bugs.

Version Control System

Version control systems are indispensable nowadays, even in solo development. Git[5] version control system was utilised throughout the project for its ease of use and elegance. Some of the benefits version control systems provide include:

- Code history. Able to go back in time
- Backup
- Maintain multiple versions of the software

Boot loader

The aim of this project is to introduce operating system concepts, therefore developing a boot loader from scratch is unnecessary and time-consuming. An open-source boot loader, GRUB[6], was chosen as a suitable boot loader for the operating system.

Chapter 3 - Design

This chapter describes the overall architecture of *Incitatus*, the proposed operating system and its components with the aid of relevant figures. Incitatus, by its design aims to be as modular, extensible and modifiable as possible in order to be an effective pedagogical tool. These requirements have greatly influenced the overall architecture of Incitatus.

Modularity in Incitatus is accomplished through the use of *kernel modules* which are explained in 3.2. Modifiability and extensibility are often interrelated as they require the operating system to be architecturally "flexible". This is achieved by utilising module interfaces which are also described in 3.2.

3.1 System Architecture

Incitatus consists of a kernel and user space applications which use facilities and services provided by the kernel. Thus the kernel, the core component of the operating system, is a layer of abstraction between the hardware and the software. The kernel itself consists of numerous interconnected components, or facilities, which:

- Control the creation, termination and communication of processes
- Manage and allocate system resources such as physical memory
- Organise and manage the file system
- Handle various system devices

Figure 3.1 outlines the general structure of Incitatus:

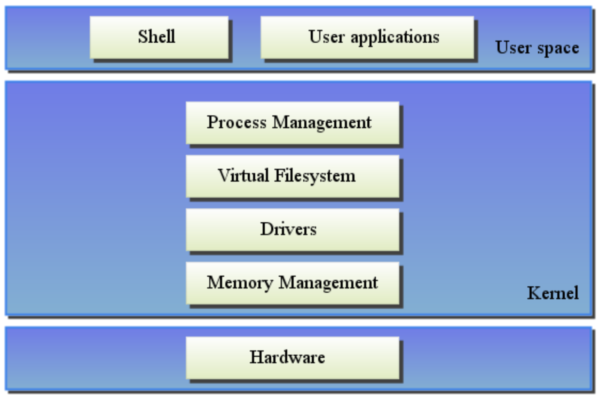


Figure 3.1: Incitatus' architecture

Incitatus' kernel is monolithic. The main reason is that I found them easier to understand and implement compared to other kernel designs such as micro-kernels. As presented in Introduction (1.3), the ease of understanding is a crucial project requirement.

In a monolithic kernel design, the whole kernel is located in the same address space as a single, large binary and all the operating system services are provided using system calls.

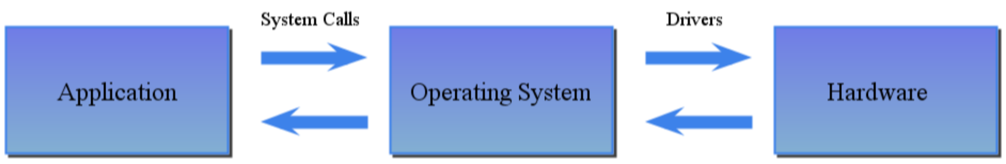


Figure 3.2: Monolithic Kernel

System calls form the interface between a user process and the kernel; providing a point of entry to operating system services. A user process must explicitly raise an interrupt, a type of signal, in order to request a service from the kernel. System calls transfer the execution from user mode to kernel mode, a privileged mode which the kernel operates in.

The actual switch between privilege modes or *protection rings* requires the co-operation of both the operating system and the hardware; where the semantics of protection rings is typically enforced by the hardware. The x86 architecture provides four protection rings numbered from 0 to 3 to be used by an operating system where a greater number means lower privilege level.[15][p. 6-6]

Incitatus makes use of ring zero and ring three. The operating system kernel operates in the most privileged mode, ring zero, which is also defined as the supervisor mode. In supervisor mode, the code may execute a number of privileged instructions that are not available in other modes such as halting the CPU. Without such protection measures, a user application may easily damage the operation of the whole operating system. User applications and therefore the Shell reside in ring three. Users interact with the operating system through the command line shell and other user applications. These applications and the shell are not part of the kernel as they are executed as separate processes. In most mainstream operating systems running on X86 architecture such as Windows[28], Linux[10][p. 50] or Mac OS X[18][p. 267], kernel mode runs in ring 0 and user mode runs in ring 3. Rings 1 and 2 were traditionally meant to be used by semi-privileged code such as device drivers, but for portability reasons (some CPU architectures only support two privilege levels) they are generally unused.

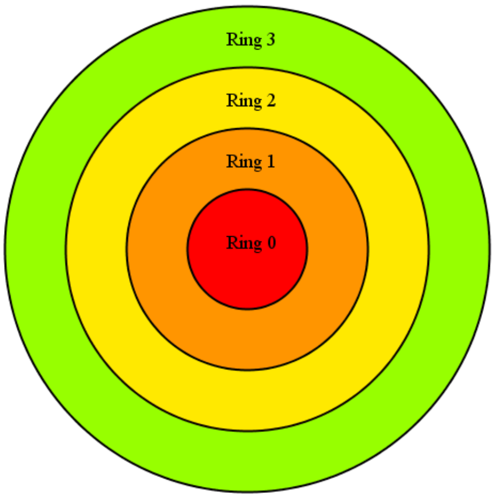


Figure 3.3: Protection rings on x86 architecture

3.2 Kernel Organisation

This section outlines the components that make up the Incitatus operating system. Some of these components act merely as a foundation for other components. Components that do not belong to that category may be viewed as service providers to user processes.

At the lowest level resides the boot-loader. The job of the boot-loader in Incitatus is to bootstrap the operating system by loading instructions from read only memory to random access memory (main memory). Following a power-up, the processor is placed in real-address mode. Real-address mode restricts the possible addressable space to one megabyte. Boot-loader switches from real-address mode to protected mode, an x86 mode of operation that as well as increasing the addressable space (to 4 gigabytes on 32 bit architecture) enables the use of security features such as virtual memory and paging[15][p. 3-1]. Boot-loader also loads the *initrd* module, the ram disk which is used as a read-only file system.

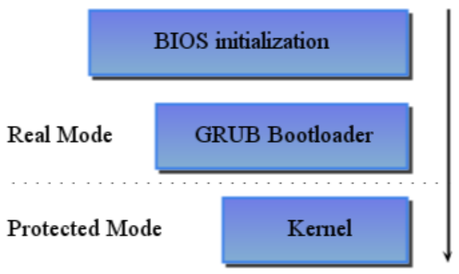


Figure 3.4: Boot process

As soon as boot-loader finishes its job, it jumps to kernel initialisation. Each component gets initialised and set up by the kernel. Components or *modules* represent the operating system primitives such as memory or process managers. Modules may depend on the functionality of other modules. The initialisation and the dependency check for each module is handled by the module manager. Following a successful (meaning all its dependencies are met) dependency check, the module is initialised.

Kernel Modules

Kernel modules make it easier to grasp the internal workings of the kernel as each critical kernel primitive is represented as a single module. Some of the modules provide implementation interfaces. For example, a student may "plug in" a different heap management implementation by implementing the interface required by the *HeapMemory* module. Each module aims to be as self-contained as possible, exposing as little as possible to other modules. This design principle is known as information hiding (or *encapsulation*) and is practised throughout the operating system. Figure 3.5 outlines the modules that make up the kernel.

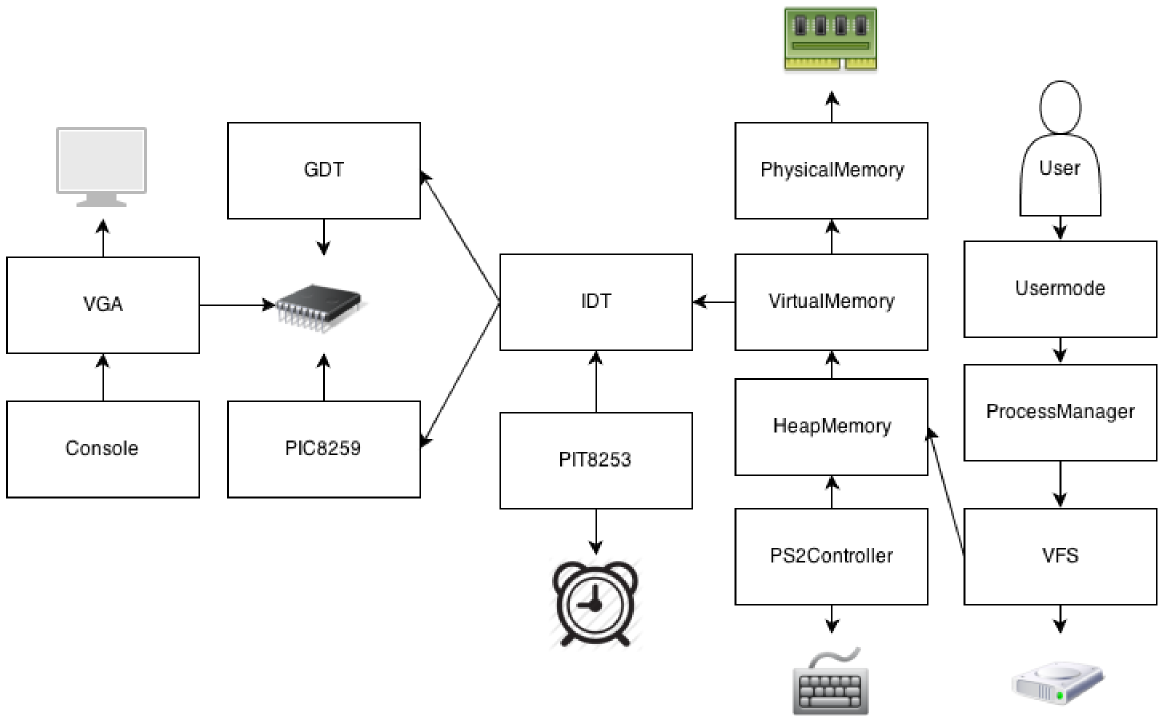


Figure 3.5: Kernel modules

VGA

The VGA module handles the video framebuffer and thus makes it possible to write text to the screen. It does not depend on any other module.

Console

The console module sits as an abstraction on top of the framebuffer and acts as the main display of the operating system. It handles scrolling, displaying, coloring and formatting of text. It depends on the *VGA* module.

GDT

The GDT module manages global descriptor table and task state segment. Task state segment is an essential structure for implementing multi-tasking support in x86 architectures. Global descriptor tables are used by the x86 architecture as data structures that define the security characteristics of an arbitrary memory area (or segment). *"When operating in protected mode, all memory accesses pass through either the global descriptor table (GDT)..."[16][p. 2-3]* Incitatus does not actively use global descriptor tables as a security detail but nevertheless the x86 architecture requires that these tables are present and valid. Global descriptor tables are set so that the memory remains flat (CPU can address all of the memory). This is called flat memory model and is practised in other operating systems such as Linux[10][p. 42] and Windows[29]. GDT module does not depend on any other module.

PIC8259

This module acts as a driver for the Intel x86, 8259 Programmable Interrupt Controller (PIC). PICs make it possible to receive hardware interrupts from external sources or devices such as hard disks, keyboards, etc.. PIC8259 does not depend on any other module.

IDT

IDT module manages the interrupt descriptor table. This module is responsible for calling the appropriate interrupt handler whenever an interrupt is raised. IDT depends on *GDT* and *PIC8259* modules.

PIT8253

This module provides an interface for the Intel 8253 Programmable Interval Timer (PIT). PIT consists of three timers (or channels), each with a different purpose. In their default states:

- First timer is used as the system timer
- Second timer is used for RAM refreshing
- Third timer is connected to PC speaker (that annoying beeper)

Incitatus only utilises the first timer which raises an interrupt on every clock tick. This timer is used for determining when to switch the process context. PIT8253 depends on the *IDT* module.

PhysicalMemory

This module provides an interface for physical memory management implementations. A valid implementation handles the allocation and deallocation of physical memory. This module does not depend on any other module.

VirtualMemory

The VirtualMemory module sets up the virtual memory through paging. Paging is used instead of segmentation (global descriptor tables) to secure and separate process address spaces. This module depends on *IDT* and *PhysicalMemory* modules.

HeapMemory

The HeapMemory module manages the kernel and user heaps. It also provides an interface for a heap management implementation. The default implementation for both kernel and user heaps is Doug Lea's malloc[17]. This module depends on the *VirtualMemory* module.

PS2Controller

This module sets up the keyboard controller. It depends on the *HeapMemory* module.

VFS

The VFS module supervises the virtual file system. It tries to hide the differences between different file system formats. Thus, Virtual file system enables the operating system to present the whole file system in a uniform manner. It depends on the *HeapMemory* module.

ProcessManager

The ProcessManager module handles the creation, destruction and resource allocation of processes. It depends on the *VFS* module.

Usermode

This module initialises the system calls list and jumps to initial user process, which is the shell. It depends on the *ProcessManager* module.

3.3 Process Management

Process management is handled by the *ProcessManager* kernel module. Incitatus supports preemptive and nonpreemptive multitasking depending on the scheduler implementation. Each task is termed a process and processes have a single thread of execution, thus multi-threading is not supported.

A process's life cycle is determined using the five-state process management model. After receiving a '*spawn* a new process' system call, *ProcessManager* module allocates resources for the process and adds it to the scheduling queue. While running, the process may get blocked; waiting for an external event to occur. The process is executed again if and only if an event occurs. As soon as a process finishes its task (ends its thread of execution), it is marked as destroyable, is removed from the scheduling queue and has its resources deallocated. Figure 3.6 outlines such a scenario.

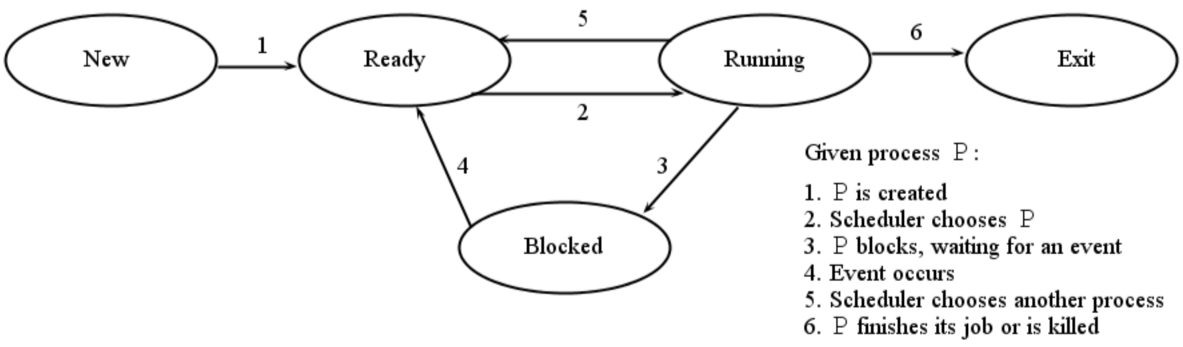

New
1
Ready
5
2
Running
6
Exit
4
3
Blocked
Given process P :
1. P is created
2. Scheduler chooses P
3. P blocks, waiting for an event
4. Event occurs
5. Scheduler chooses another process
6. P finishes its job or is killed

Figure 3.6: Five-state process model

Incitatus provides a scheduler interface for process scheduler implementations. This interface makes it possible to implement preemptive and nonpreemptive scheduling algorithms. Incitatus includes two scheduler implementations:

1. Preemptive round-robin scheduling

2. Nonpreemptive first come, first served scheduling

In Incitatus, a process may operate in either user mode or kernel mode. While executing the contents of a binary, a process is in user mode. Whenever a system call is made, the same process jumps to the kernel mode and starts to execute code inside the kernel. For this reason, a process has different resource requirements for user mode and kernel mode. Such details are given in the Implementation (4) chapter.

3.4 Memory Management

Memory management in Incitatus is handled by three kernel modules, each controlling a different level of abstraction:

- *PhysicalMemory* module deals with the lowest level of abstraction, the physical memory.
- *VirtualMemory* module takes care of virtual memory.
- *HeapMemory* module handles kernel and user heaps.

PhysicalMemory and *HeapMemory* modules provide implementation interfaces. Incitatus comes with two different physical memory manager implementations, a bitmap (or bit array) based and a stack based. They each have their own strengths and weaknesses, which are discussed in the Implementation (4) chapter. The heap manager implementation for both kernel space and user space is a port of Doug Lea's Malloc[17]. Doug Lea's Malloc is a well known general purpose memory allocator that aims to balance memory wastage and performance.

VirtualMemory module does not provide any interfaces as it is platform dependent (X86 architecture provides hardware support for managing virtual memory with paging). Each process in Incitatus has its own independent address space, managed by the virtual memory manager. This means that processes can not accidentally or intentionally access or manipulate an address belonging to another process' address space. If an invalid request is made to a memory address by a process executing in user mode, a page fault is raised to kill the faulting process.

3.5 File System

Operations concerning the file system is managed by the *VFS* module in Incitatus. As shown in figure 3.7, the virtual file system acts as an abstraction in order to provide a simple file interface to processes. When a process requests a service (such as opening or reading from a file) concerning a file, virtual file system manages all the details such as communicating with the file system architecture and the device driver.

For example, let's assume we have a file that we want to access such that:

- The file is stored inside a SATA hard disk.
- The file system architecture containing the file is FAT32.

When a request is made, virtual file system uses the low-level FAT32 file system driver and the SATA hard disk driver to reach and interact with the file.

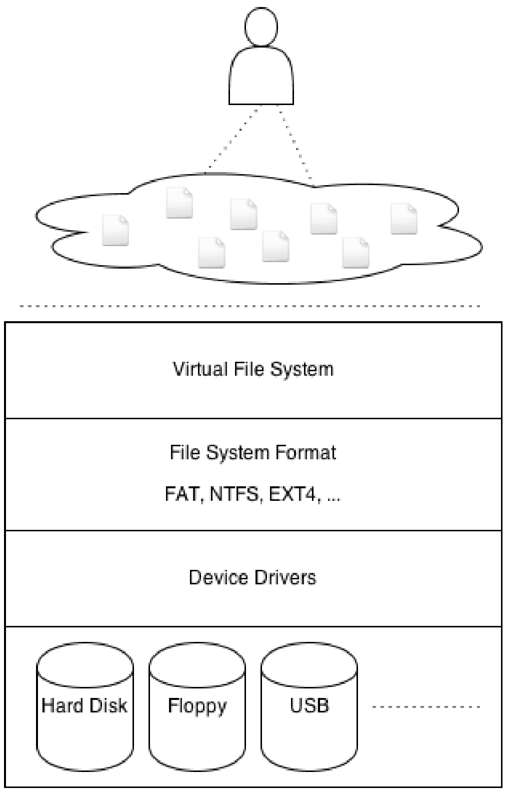


Figure 3.7: Virtual file system organisation

Incitatus does not provide drivers for either SATA or FAT32. Instead, Incitatus' whole file system is stored in the primary memory, also known as a ram disk. The file system architecture for the ram disk is TAR[26] (as in TAR archive). TAR archives contain meta information such as file name, file path, file type and file size. The only task that is left for the file system is to parse that information and this is exactly what Incitatus does.

Chapter 4 - Implementation

This section explains how the Incitatus operating system is implemented. Operating systems development, by its nature, involves low-level implementation details. Such details are given when it is absolutely relevant to the topic being presented.

In Incitatus, modularity is achieved through the use of kernel modules; some of these modules also define interfaces which allow students to modify the functionality of the operating system. Sample implementations are provided for all of these modules which will be described in this chapter.

All the interfaces provided by Incitatus are also presented in this chapter. C programming language does not provide any syntax support for defining interfaces as opposed to many other programming languages such as Java. Fortunately, function pointers make it possible to create interfaces in C and they are used in Incitatus to represent the module interface functions.

The source code of the operating system has been extensively commented to ease its modification. Chapter 5 describes how to modify the operating system in detail.

4.1 Booting and Kernel Initialisation

Booting process in Incitatus is handled by *GRand Unified Bootloader* (GRUB[6]) which is a Multiboot[12] compliant bootloader. Multiboot specification ensures that a multiboot compliant kernel can be booted by any multiboot compliant bootloader (in Incitatus' case, GRUB).

As soon as GRUB finishes its job, it leaves the system in the following state and transfers the execution of the system to the kernel:

- CPU is in protected mode (A20 gate is enabled).

- Interrupts are disabled.
- Paging is disabled.
- EAX register contains multiboot magic value.
- EBX register contains pointer to multiboot information structure.
- ESP (stack) register is undefined.

The first job of the kernel is setting up its own stack by defining the ESP register:

Listing 4.1: Start.s

```
mov esp, stack + STACKSIZE ; STACKSIZE is defined as 64 kilobytes
```

Then, it checks for the multiboot information contained in the EAX and EBX registers to ensure that the boot process went successfully (lines 5-7 in listing 4.2). After that, the module manager begins to load the kernel modules (lines 9-28 in listing 4.2):

Listing 4.2: Kernel.c

```
1  PUBLIC void Kernel(MultibootInfo* mbInfo, u32int mbMagic) {
2
3          ...
4
5      Debug_assert(mbMagic == MULTIBOOT_BOOTLOADER_MAGIC);
6      Debug_assert(mbHead.magic == MULTIBOOT_HEADER_MAGIC);
7      Debug_assert(mbInfo->modsCount > 0); /* Make sure initrd(ram disk) is in memory */
8
9      Module* modules[] = {
10
11         VGA_getModule(),
12         Console_getModule(),
13         GDT_getModule(),
14         PIC8259_getModule(),
15         IDT_getModule(),
16         PIT8253_getModule(),
17         PhysicalMemory_getModule(),
18         VirtualMemory_getModule(),
19         HeapMemory_getModule(),
20         PS2Controller_getModule(),
21         VFS_getModule(),
22         ProcessManager_getModule(),
23         Usermode_getModule(),
24
```

```
    };

    for(u32int i = 0; i < ARRAY_SIZE(modules); i++)
        Module_load(modules[i]);

        ...

}
```

4.2 Memory Management

Memory management is a major component in Incitatus; almost every kernel module makes use of its functionalities one way or another. Memory management is layered and implemented in a hierarchical fashion in order to:

- Separate architecture-dependant and architecture-independent code.
- Maintain modularity.
- Enhance modifiability and extendibility

4.2.1 Physical Memory

As presented in 3.4, physical memory (primary memory, the RAM) is handled by the *PhysicalMemory* kernel module. This module provides the following interface:

Listing 4.3: PhysicalMemory.h

```
extern void (*PhysicalMemory_init) (MultibootInfo* mbI, MultibootHeader* mbH);
extern PhysicalMemoryInfo* (*PhysicalMemory_getInfo) (PhysicalMemoryInfo* buf);
extern void* (*PhysicalMemory_allocateFrame) (void);
extern void (*PhysicalMemory_freeFrame) (void* frame);
```

The interface consists of an initialisation function (*PhysicalMemory_init*) that prepares the physical memory by gathering how much memory the system has (by analysing the multiboot structures) and marking sections of memory that are already reserved (such as the kernel itself). Valid implementations must also implement functions that allocate (*PhysicalMemory_allocateFrame*) and free (*PhysicalMemory_freeFrame*) frames as well as a function (*PhysicalMemory_getInfo*) that returns information on the current state of physical memory. Frames are four kilobytes of contiguous physical memory blocks. The reasoning behind this is given in 4.2.2.

Incitatus provides bitmap-based and stack-based implementations of the *PhysicalMemory* interface to demonstrate the modifiability of the OS.

Bitmap-based

Bitmap based physical memory manager makes use of a bit array data structure in order to record the state of all the frames. A frame is either free or allocated, which means a single bit is enough to store such information. The array index of a bit specifies the state of the nth frame, this implies that the state of every single frame is known.

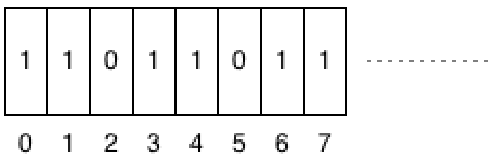


Figure 4.1: Frame states bit array

Stack-based

Stack based physical memory manager, as its name suggests, takes advantage of the stack data structure. Free frames are pushed onto stack and newly allocated frames are popped off the stack. Unlike the bitmap based approach the state of a given frame is not known, as only the top of the stack is visible.

Comparison

Bit array is capable of storing the state of eight frames per byte (assuming one byte is eight bits). Whereas stack needs four bytes (in 32-bit architecture) in order to store the address of a free frame. Allocation in bitmap-based manager is slower as it needs to iterate through the array until it finds the first free frame. On the contrary, stack-based manager does not need to search for a free frame as it only stores free frames. Table 4.1 outlines the performance of these two physical memory manager implementations.

	Bitmap	Stack
Free frame	$O(1)$	$O(1)$
Allocate frame	$O(n)$	$O(1)$
Space	$O(n/8)$	$O(4n)$

Table 4.1: Bitmap versus stack physical memory managers

4.2.2 Virtual Memory

Incitatus utilises virtual memory in order to provide security and the illusion of contiguous memory to processes. X86 architecture provides hardware support for virtual memory through paging[16][p. 4-1]. Paging involves several data structures that map the virtual memory to physical memory.

Incitatus is a 32-bit operating system therefore the size of the virtual memory address space is 4 gigabytes. Regardless of how much physical memory the system has, the only address space processes see is the virtual address space (once paging is enabled). Paging needs to be enabled before it can be utilised by manipulating the *CR0* (Control Register) register and specifying the page directory in *CR3* register. Figure 4.2 shows the address space organisation in Incitatus.

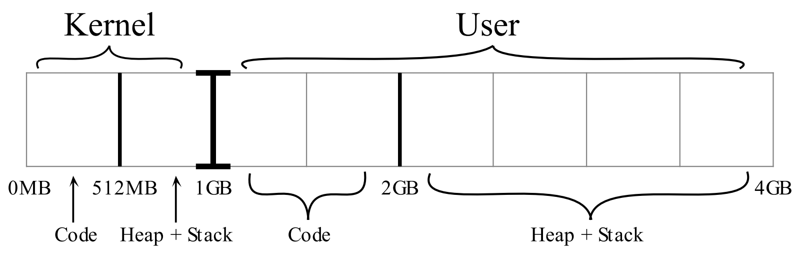


Figure 4.2: A process' address space in Incitatus

Paging Structures

Incitatus defines four types of paging structures:

- Page Table Entry represents a single page. A page is a four-kilobyte virtual memory block which is mapped to a four-kilobyte physical memory block.
- Page Table is an array of page table entries.
- Page Directory Entry represents a single page table.
- Page Directory is an array of page directory entries.

In the lowest level is the Page Table Entry (PTE). PTEs manage four-kilobyte frames. These physical frames need to be mapped inside the PTE structure before they can be accessed through virtual memory. Listing 4.4 shows the PTE structure.

Listing 4.4: VirtualMemory.c

```
/* 4-Byte Page Table Entry */
struct PageTableEntry {

    u8int inMemory : 1; /* Whether the page is in memory(is present) or not */
    u8int rwFlag : 1; /* Read/Write flag */
    u8int mode : 1; /* Page operation mode. 0: kernel mode, 1: user mode. */
    u8int : 2; /* Reserved */
    u8int isAccessed : 1; /* Access flag */
    u8int isDirty : 1; /* Has this page been written to? */
    u16int : 2; /* Reserved */
    u8int : 3; /* Available for use */
    u32int frameIndex : 20; /* Frame address */

} __attribute__((packed));
```

Page tables are essentially arrays containing 1024 PTEs. Thus, each page table manages $1024 \times 4KB = 4MB$ of virtual memory.

Listing 4.5: VirtualMemory.c

```
struct PageTable {

    /* 4MB of virtual memory */
    PageTableEntry entries[1024];

};
```

Page directory entries (PDE) are pretty similar to PTEs, except the fact that they point to page tables instead of page frames. X86 architecture allows us to specify the page size. Depending on how the PDEs are set up, the page size is either four megabytes or four kilobytes. Incitatus uses four-kilobyte sized pages and this is why the physical memory is managed with four-kilobyte blocks.

Page directories are arrays of 1024 PDEs. So, each of them manages $1024 \times 4MB = 4GB$ of virtual memory. A page directory, as a matter of fact, can be considered as the whole address space of an individual process.

Listing 4.6: VirtualMemory.c

```
struct PageDirectory {

    /* 4GB of virtual memory */
    PageDirectoryEntry entries[1024];

};
```

Security

Perhaps the most crucial benefit of paging is the fact that we can set the privilege level of individual pages. In Incitatus, all kernel pages (the first gigabyte) are set as *supervisor* pages. Thus, user space code can not access pages in the kernel space. Whereas kernel space code can access all of them.

Page Fault

Attempting an access to a non-mapped or a privileged page results in a *page fault*. Page fault handling in Incitatus is relatively basic; the faulting process is killed if it is running in user mode. If the page fault is raised while executing in the kernel mode, kernel is halted (kernel panic).

MMU and TLB

Memory Management Unit is the hardware unit that manages the virtual memory by translating virtual addresses to physical addresses with the help of paging structures. MMU includes a *translation lookaside buffer* (TLB) that handles the caching of individual pages, improving the translation speed. *VirtualMemory* module provides two functions that manipulate the TLB:

Listing 4.7: VirtualMemory.c

```
PRIVATE inline void VirtualMemory_invalidateTLB(void);
PRIVATE inline void VirtualMemory_invalidateTLBEntry(void* addr);
```

The first one invalidates (deletes) all the entries inside the TLB. Whereas the second one invalidates a single entry.

A virtual address (or linear address) is made of three segments in Incitatus. MMU ”knows” which virtual address is mapped to a physical one. While in translation process, MMU uses these segments to locate the correct page table entry, after locating the page table entry it simply gets the mapped physical address. Figure

4.3 shows these segments and the overall structure of paging in Incitatus.

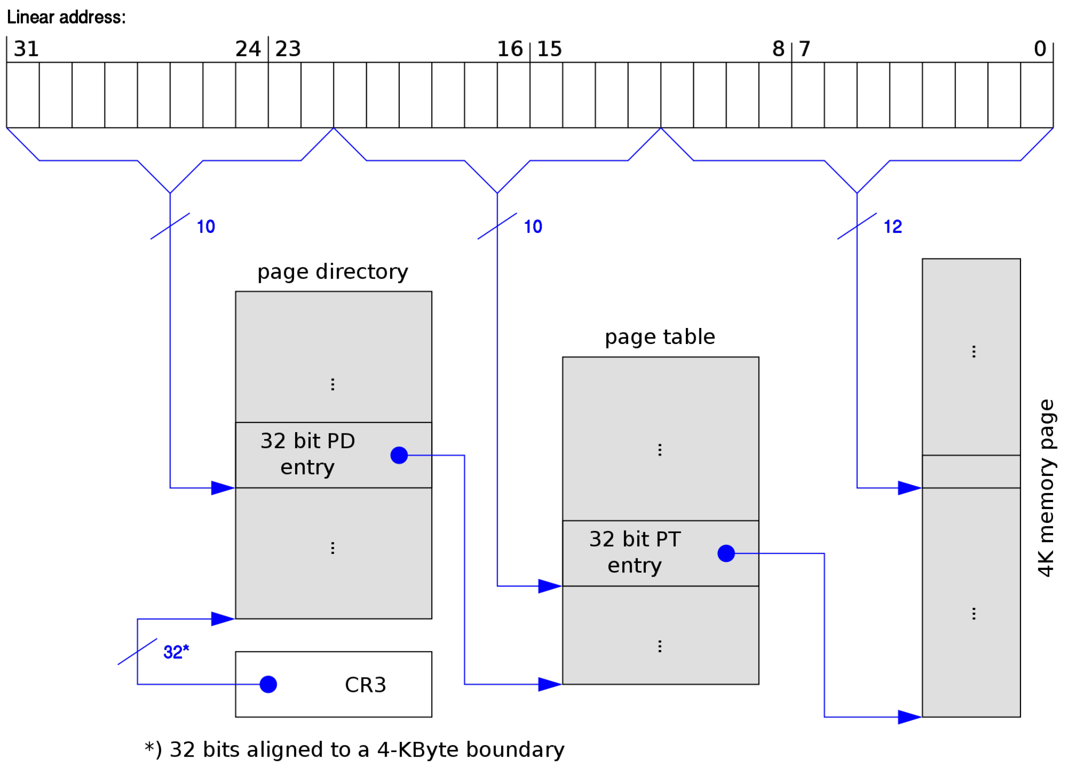


Figure 4.3: Paging structures in Incitatus (Wikipedia[27])

4.2.3 Heap Memory

HeapMemory module provides the following interface for heap memory manager implementations:

Listing 4.8: HeapMemory.h

```
/* Allocates "bytes" number of bytes of space in heap and returns the
 * allocated address. */
extern void* (*HeapMemory_alloc) (size_t bytes);

/* Reallocates the given memory block to a new block of size "bytes". */
extern void* (*HeapMemory_realloc) (void* oldmem, size_t bytes);

/* Allocates an array of "numberOfElements" elements and sets all bits
 * as zero. Each element has "elementSize" length. */
extern void* (*HeapMemory_calloc) (size_t numberOfElements, size_t elementSize);
```

```
/* Deallocates the specified memory block. */
extern void (*HeapMemory_free) (void* mem);
```

Kernel and the user space both have separate heaps which can "expand" or "contract" if required. In kernel space, this is accomplished by the following function:

Listing 4.9: HeapMemory.h

```
/* Expands or contracts(if negative value given) the kernel heap
 * space by "size" bytes. "size" needs to be page aligned. */
void* HeapMemory_expand(ptrdiff_t size);
```

Kernel heap manager implementations must make use of this function in order to implement the interface. The user space equivalent of this function is the *sbrk* system call.

The default heap manager implementation for both the kernel and the user space is Doug Lea's Malloc (dlmalloc). Dlmalloc manages the heap using *bins* (or buckets) which are chunks of memory sorted according to their size. Searching for a free chunk (allocation) is performed using the best-fit strategy, which tries to find the smallest bin that can hold the chunk. Freed chunks are coalesced (merged) with neighbouring free chunks, thus effectively reducing external memory fragmentation. If dlmalloc runs out of free bins, it requests more memory from the operating system (heap "expansion"). Dlmalloc may also shrink the heap by returning back memory to the operating system (heap "contraction") if unused heap is above some certain threshold; which is two megabytes on default settings.

4.3 File System

Incitatus contains a hierarchical file system implementation (the ramdisk) which is the traditional way of representing files based on a hierarchy determined by the ordering of files, directories and subdirectories.

Virtual File System

Virtual file system defines an interface for the actual file system implementations. Contents of files are managed by these file system implementations, thus a file system can be considered as a driver. Listing 4.10 shows the structure of a file system. The function pointers contained inside that structure represent the interface used by the file nodes to interact with file objects.

Listing 4.10: VFS.h

```
struct VFS {
    u32int deviceID;
    VFSNode* rootNode;

    /* Opens a file and returns it as a file descriptor(node pointer). */
    VFSNode* (*open) (VFSNode* self);

    /* Closes an opened file. */
    VFSNode* (*close) (VFSNode* self);

    /* Reads from a file. */
    u32int (*read) (VFSNode* self, u32int offset, u32int count, char* buffer);

    /* Writes to a file. */
    u32int (*write) (VFSNode* self, u32int offset, u32int count, const char* buffer);

    /* Returns nth child of a directory. */
    VFSNode* (*readDir) (VFSNode* self, u32int index);

    /* Finds child file in a directory and returns it as a file descriptor */
    VFSNode* (*findDir) (VFSNode* self, const char* path);
};
```

Virtual file system revolves around the concept of *nodes* (similar to UNIX' *inode*). Each node stores all the information of an individual file object. Listing 4.11 shows the layout of a virtual file node. Each node "knows" which file system it belongs to. Thus, a node is able to use the interface provided by its host file system to interact with the file object.

Listing 4.11: VFS.h

```
struct VFSNode {

    char fileName[VFS_FILE_NAME_SIZE]; /* The file name */
    u32int index; /* Index of the node in a specific device*/
    u32int permission; /* rwx Permission mask similar to Unix systems (for example 644, 777, etc.) */
    u32int mode; /* 0 = file is not open, 1 = read mode, 2 = write mode */
    u32int uid; /* Owner id */
    u32int gid; /* Group id */
    u32int fileType; /* Directory, normal, link, etc. */
    u32int fileSize; /* Size of the file in bytes */
    VFS* vfs; /* File system this node belongs to */
    VFSNode* ptr; /* Used by mountpoints and symlinks */

};
```

Ramdisk

The only "concrete" file system implementation in Incitatus is a parser for tar archives. Tar format was chosen because of its simplicity and widespread use. A tar archive is essentially a linked list of tar "entries". Each tar entry is made up of a header and its corresponding file contents. The header stores meta information about an individual file such as file name, file size and file type. Listing 4.12 shows the complete structure of a tar header.

Listing 4.12: Tar.h

```
struct TarEntryHeader {

    char fileName [TAR_FILE_NAME_SIZE];
    char fileMode [TAR_FILE_MODE_SIZE];
    char userID [TAR_UID_SIZE];
    char groupID [TAR_GID_SIZE];
    char fileSize [TAR_FILE_SIZE_SIZE];
    char lastModified [TAR_LAST_MODIFIED_SIZE];
    char checksum [TAR_CHECKSUM_SIZE];
    char fileType [TAR_TYPEFLAG_SIZE];
    char linkName [TAR_LINKNAME_SIZE];

};
```

While still in the boot process, the boot-loader puts the whole tar archive inside the main memory, just after the end of the kernel. The *VFS* module then initialises the ramdisk; which retrieves the location of the tar archive and parses it, building a list of files.

Incitatus' root file system is the ramdisk (as it is the only file system implementation), meaning all other file systems are mounted on the ramdisk. Mounting another file system requires the root file system to be writable. Unfortunately, the ram disk driver in Incitatus is read only (because of time constraints). Thus, operations such as mounting a file system, writing to a file or creating a new file are not possible. Incitatus is capable of using any TAR archive as its file system. A host operating system may create a non-empty TAR archive and pass it to the compilation script.

4.4 Process Management

4.4.1 Processes

Each process has its own address space, user heap, user space stack, kernel space stack, and a list of opened files in Incitatus. Having separate stacks is necessary as system calls, naturally, should not be using the user stack because of security implications. All process structures contain a page directory, which form its entire address space. Listing 4.13 shows the internal structure of a process.

Listing 4.13: ProcessManager.h

```
struct Process {
    u32int pid;
    u8int status;
    char name[64];
    void* userHeapTop;
    void* kernelStack;
    void* kernelStackBase;
    void* userStack;
    void* userStackBase;
    void* pageDir;

    VFSNode* workingDirectory;
    ArrayList* fileNodes;
};
```

During the creation of a new process, kernel code and heap is mapped to a newly created address space. This is necessary as they need to be accessed following a system call. Binary file contents (the code) are also mapped to user code virtual address which starts at one gigabyte (4.2.2).

There are two kinds of processes in Incitatus: kernel processes and user processes. The only kernel process is an idle process. This process repeatedly halts the CPU, reducing the amount of wasted cycles. Without such an idle process, the CPU would always execute instructions (busy wait) even if all the user processes were blocked.

Inter-process Communication

Processes in Incitatus communicate with each other using a relatively simple IPC implementation. All messages between processes are exchanged through a *global mailbox*. The mailbox itself is simply an *ArrayList* of messages. Listing 4.14 shows the structure of messages.

Listing 4.14: ProcessManager.c

```
PRIVATE ArrayList* globalMailbox;
...
struct Message {
    Process* to;
    Process* from;
    u32int message;
};
```

For example, the *waitpid* system call sends a message from the calling process A to the specified (argument) process B, informing B that A is waiting for the end of Bs execution. This is shown in listing 4.15.

Listing 4.15: ProcessManager.c

```
Message* m = HeapMemory_calloc(1, sizeof(Message));
m->from = Scheduler_getCurrentProcess();
m->to = process;
m->message = PROCESS_MSG_WAITING;
ArrayList_add(globalMailbox, m);
ProcessManager_blockCurrentProcess();
```

Context Switch

An interrupt handler is responsible for switching processes whenever a system timer interrupt is generated. Context switch involves saving the state of the current process, selecting the next process from the scheduling list and switching to the next process' address space. Context switch occurs only if the next process in the scheduling list is not actually the currently running process. If it is, the interrupt handler simply returns without initiating a context switch.

4.4.2 Scheduling

Incitatus provides the following interface for scheduler implementations:

Listing 4.16: Scheduler.h

```
/* Adds a new process to scheduler's process list. */
extern void (*Scheduler_addProcess) (Process* process);

/* Removes a process from scheduler's process list. */
extern void (*Scheduler_removeProcess) (Process* process);

/* Returns the next process to be switched to. */
extern Process* (*Scheduler_getNextProcess) (void);

/* Returns the current process. */
extern Process* (*Scheduler_getCurrentProcess) (void);
```

Round-robin and first come, first served scheduler implementations are provided as an example, and to demonstrate the modularity of the operating system.

Round-robin Scheduling

Round-robin scheduling is a preemptive scheduling algorithm. In Incitatus, round-robin scheduler implementation stores all the processes (be it blocked, waiting, running, etc.) in a linked list data structure. Whenever a context switch interrupt is raised, the round-robin scheduler selects a waiting process from the list. Round-robin scheduler adds new processes to the end of the linked list.

FCFS Scheduling

First come, first served (FCFS) is a non-preemptive scheduling algorithm. Similar to round-robin, FCFS scheduling implementation stores all the processes inside a linked list data structure. The same process is selected every time a context switch interrupt is raised. As soon as the running process ends its execution, the next one in the list is selected. FCFS scheduler also adds new processes to the end of list.

4.5 User Space

The first user application that is executed soon after jumping to user space is the shell. Shell provides a command line interface, acting as a gateway between the operating system and the user. Other applications may be executed with a specific command provided by the shell. Commands provided by the shell are given in table 4.2. The shell and other user applications communicate with the kernel using a set of POSIX-inspired[14] system calls. Table 4.3 outlines the system calls that can be used by user space applications.

Command	**Description**
ls	list files inside current working directory
cd [dir]	change working directory
cls	clear console
cat [file]	display file contents
restart	restart machine
exec [file]	execute binary file
suicide	kill the shell
shutdown	shut down the machine

Table 4.2: Shell Commands

User space applications are compiled and linked (thoroughly explained in chapter 5) using a static user space library. The library provides system call functions and acts as a minimal C library. Most of the C library was borrowed (permitted under the MIT license) from *musl*[9] as even a minimal libc implementation is a project in its own right.

Incitatus' *libc* consists of the following headers:

limits.h Defines the size limits of 32-bit variables.

stdint.h Defines pointer types.

stdio.h Defines I/O related functions such as *printf*.

stdlib.h Defines dynamic memory allocation functions (Doug Lea's Malloc).

string.h Defines string related functions.

File related functions are provided by the system call library.

Index	System Call	Description	Arguments	Return
0	puts	Prints a null terminated string.	Null terminated string pointer	N/A
1	putc	Prints an ASCII character.	Character to print	N/A
2	exit	Terminates the current process.	The exit code	N/A
3	spawn	Spawns a new process from an executable binary file.	The binary pathname	Process Descriptor
4	fchdir	Changes the current process' working directory.	Destination file descriptor	Destination file descriptor
5	fparent	Returns the parent directory descriptor.	The child file descriptor	Parent directory descriptor
6	getcwd	Returns the current process' working directory.	Buffer to store working directory path	The buffer containing the working directory
7	readdir	Return nth child of a directory.	Directory file descriptor Index of child	Nth child file descriptor
8	fgetcwd	Returns the current process' working directory.	N/A	Working directory file descriptor
9	fstat	Fills the specified buffer with file information.	File to be examined Buffer to be filled	N/A
10	getch	Reads and returns a character stored inside key buffer.	N/A	Character value
11	cls	Removes all characters from console screen.	N/A	N/A
12	restart	Restarts the machine.	N/A	N/A
13	open	Opens a file.	Path of the file to be opened Read-write mode	A file descriptor
14	close	Closes an opened file.	File descriptor	0 if file closed successfully
15	read	Reads from a file.	The file to read from Offset Number of bytes to read Buffer to store read bytes	The number of read bytes
16	write	Not implemented	N/A	N/A
17	mkdir	Not implemented	N/A	N/A
18	sbrk	Expands or contracts the user heap.	Page aligned number of bytes	Pointer to heap top
19	finddir	Finds child file in a directory.	Parent directory descriptor Absolute child path	Child file descriptor
20	chdir	Changes the current process' working directory.	Destination path	Destination file descriptor
21	color	Sets the text and background color.	Color to set	N/A
22	poweroff	Shutdown machine.	N/A	N/A
23	waitpid	Wait for specified process' termination.	The process (descriptor) to wait for	N/A

Table 4.3: System Calls

4.6 Test Environment

Kernels are not as straightforward to test as user space applications. This is largely because of the fact that we deal with the actual hardware in most of the cases. Thus, we either have the option to emulate the hardware in order to test it or to actually run it on the hardware. Before we choose, certain key factors have to be taken into account:

Development speed
The time it takes to deploy and test the software.

Ease of use
How easy it is to deploy new versions of the software.

Closeness to actual hardware
How similar the emulator behaves in relation to the target hardware.

The greatest advantage of emulators is the testing speed. Every iteration of the operating system needs to be tested as to get rid of software bugs as soon as possible. If we were to try this on actual hardware, we would have to burn the operating system image and reboot the computer for every iteration; which is extremely inefficient. Considering the trade-offs, the clear choice is to use a hardware emulator in order to test the operating system. While developing Incitatus, emulators QEMU, Bochs and VirtualBox have been utilised.

Chapter 5 - Manual and Example Instructional Usage

This chapter describes how to build Incitatus' source code and provides examples on how to modify it.

5.1 Requirements

The following list outlines the software necessary to build and run the operating system (on an emulator).

- A Linux distribution
- NASM x86 assembler
- GCC compiler
- x86 emulator such as QEMU, Bochs or VirtualBox

Note that a cross compiler tool-chain (GCC and GNU Binutils) is required if the host operating system is 64 bits. The link below gives a step-by-step guide on how to create a cross compiler toolchain.

`http://wiki.osdev.org/GCC_Cross-Compiler`

5.2 Source Code Structure

Figure 5.1 shows the source code structure of Incitatus. The layout of the source tree matches the overall design of the system. The bulk of the code comments are inside the header files (files with the ".h" extension) as header files are meant for users of the code.

```
/
├── user/ ............................ Contains the user-space source code
├── kernel/ ......................... Contains the kernel-space source code
├── bootloader/ .... Contains the grub bootloader and ramdisk TAR archive
├── bin/ ........................... Holds the image file and other binaries
├── run_qemu.sh .................... Runs Incitatus on the QEMU emulator
├── run_bochs.sh .................... Runs Incitatus on the Bochs emulator
├── mk.sh ............................................ Builds Incitatus
└── bochsrc.txt .................................. Bochs configuration file
```

Figure 5.1: Incitatus source code structure

5.3 Build

A shell script is included (”mk.sh”) along with the source code that builds Incitatus. Building Incitatus involves compiling and linking both the kernel space and user space source code as well as producing a bootable cd image. The build script is extensively commented in order to make its modification straightforward.

Kernel-space and user-space objects (i.e. compiled source code) are linked separately. Kernel-space code is appropriately linked so that it is placed in address $0x100000$ onwards (bootloader jumps to $0x100000$ after its job is finished) and the user-space code is linked to virtual address $0x40000000$ (while entering the user-space the kernel jumps to address $0x40000000$).

5.4 Deploy

After building Incitatus, a CD image file containing the operating system and its whole file system (the ramdisk, TAR archive) is placed inside the ”bin” folder. Scripts are provided for running the operating system on either QEMU (”run_qemu.sh”) or Bochs (”run_bochs.sh”) emulators. Incitatus may also be run using VirtualBox by simply creating a new virtual machine and mounting the CD image (with no need to create a virtual hard disk as the whole file system is stored on the main memory).

5.5 Example Instructional Usage

Module Modification

As an example, let's modify the *PIT8253* kernel module which is responsible for handling the system timer. By default, the process context switch frequency in Incitatus is 20 milliseconds. Let's change it to 100 milliseconds:

1. The context switch occurs following a system timer interrupt. Thus, the code calling the process context switch is located inside the system timer interrupt handler:

   Listing 5.1: PIT8253.c

   ```
   1  PRIVATE void PIT8253_timerHandler(Regs* regs) {
   2      tick++;
   3      ...
   4      if(tick % 2 == 0) /* context switch every 20ms */
   5          ProcessManager_switch(regs);
   6      ...
   7  }
   ```

2. Modify line 4 (knowing that every tick is 10 milliseconds apart, which also can be changed inside the same source file):

   Listing 5.2: PIT8253.c

   ```
   1  PRIVATE void PIT8253_timerHandler(Regs* regs) {
   2      tick++;
   3      ...
   4      if(tick % 10 == 0) /* context switch every 100ms */
   5          ProcessManager_switch(regs);
   6      ...
   7  }
   ```

3. Build the source code by executing the "mk.sh" script and run the modified OS on either QEMU ("run_qemu.sh") or Bochs ("run_bochs.sh").

4. Observe the changes caused by lowering the context switch frequency.

Interface Implementation

PhysicalMemory (”PhysicalMemory.h”), *HeapMemory* (”HeapMemory.h”), *VFS* (”VFS.h”) and *ProcessManager* (”Scheduler.h” defines the process scheduler interface) kernel modules all provide interfaces for implementations. Let’s implement a new heap memory manager for the kernel:

1. In order to create a heap memory manager, we need to implement the interface provided by the *HeapMemory* module:

   Listing 5.3: HeapMemory.h

   ```
   /* Allocates "bytes" number of bytes of space in heap and returns the
    * allocated address. */
   extern void* (*HeapMemory_alloc) (size_t bytes);

   /* Reallocates the given memory block to a new block of size "bytes". */
   extern void* (*HeapMemory_realloc) (void* oldmem, size_t bytes);

   /* Allocates an array of "numberOfElements" elements and sets all bits
    * as zero. Each element has "elementSize" length. */
   extern void* (*HeapMemory_calloc) (size_t numberOfElements, size_t elementSize);

   /* Deallocates the specified memory block. */
   extern void (*HeapMemory_free) (void* mem);
   ```

2. Create ”MyHeapManager.h” header file inside ”/kernel/include/Memory/” (in order to be consistent with the present source code structure), exposing the functions of your implementation:

   Listing 5.4: MyHeapManager.h

   ```
   void* MyHeapManager_alloc(size_t bytes);
   void* MyHeapManager_realloc(void* oldmem, size_t bytes);
   void* MyHeapManager_calloc(size_t numberOfElements, size_t elementSize);
   void MyHeapManager_free(void* mem);
   ```

3. Create ”MyHeapManager.c” file inside ”/kernel/src/Memory/” and implement the interface. While implementing the interface, you will most likely need to expand or contract the heap. This function is already provided in Incitatus as *HeapMemory_expand()*.

Listing 5.5: MyHeapManager.c

```
#include <Memory/MyHeapManager.h>
#include <Memory/HeapMemory.h> /* Provides HeapMemory_expand() */

PUBLIC void* MyHeapManager_alloc(size_t bytes) {
/* Implement */
}

PUBLIC void* MyHeapManager_realloc(void* oldmem, size_t bytes) {
/* Implement */
}

PUBLIC void* MyHeapManager_calloc(size_t numberOfElements, size_t elementSize) {
/* Implement */
}

PUBLIC void MyHeapManager_free(void* mem) {
/* Implement */
}
```

4. "Plug-in" your implementation by modifying the "HeapMemory.c" file:

Listing 5.6: HeapMemory.c

```
...
/* Include Heap manager implementation */
#include <Memory/MyHeapManager.h>
...
PRIVATE void HeapMemory_init(void) {
    ...
    /* Point to heap manager implementation */
    HeapMemory_alloc = MyHeapManager_alloc;
    HeapMemory_realloc = MyHeapManager_realloc;
    HeapMemory_calloc = MyHeapManager_calloc;
    HeapMemory_free = MyHeapManager_free;
}
```

5. Modify the "mk.sh" script in order to compile and link your implementation (you may easily figure out how to do this by reading the comments).

6. Build the source code by executing the "mk.sh" script and run the modified OS on either QEMU ("run_qemu.sh") or Bochs ("run_bochs.sh").

Chapter 6 - Results & Evaluation

This chapter presents the outcomes of the project. These outcomes are, then, evaluated on the basis of how well they have achieved the initial objectives.

6.1 Results

The design and implementation of Incitatus operating system was greatly influenced by the project requirements, which are mainly:

- Modularity
- Extendibility
- Modifiability

These requirements were found necessary for a healthy pedagogical environment as they all, when put together, provide the ease needed for learning OS concepts. This section presents the achieved result for each requirement.

6.1.1 Modularity

Modularity requirement was achieved by dividing the functionality of the operating system into carefully designed components (or *kernel modules*). Each module is designed to be as self contained as possible, encapsulating the private data structures and variables and exposing them to other modules if necessary. Removing modules from the operating system effectively alters its functionality. For example, removing the module which manages the virtual memory (*VirtualMemory* module) will produce an operating system capable of interacting with the physical memory without any intermediary address translation; but we would have to sacrifice the other modules which depend on its functionality (as they won't pass the dependency check).

6.1.2 Modifiability

Modifiability requirement was achieved through extensive code comments and a special attention to code readability. To put the importance of readability into context, listings 6.1 and 6.2 provide context-switch code of linux v0.01 and Incitatus.

Listing 6.1: Linux 0.01, sched.c

```
void schedule(void)
{
        int i,next,c;
        struct task_struct ** p;

/* check alarm, wake up any interruptible tasks that have got a signal */
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if (*p) {
                        if ((*p)->alarm && (*p)->alarm < jiffies) {
                                        (*p)->signal |= (1<<(SIGALRM-1));
                                        (*p)->alarm = 0;
                                }
                        if ((*p)->signal && (*p)->state==TASK_INTERRUPTIBLE)
                                (*p)->state=TASK_RUNNING;
                }

/* this is the scheduler proper: */
        while (1) {
                c = -1;
                next = 0;
                i = NR_TASKS;
                p = &task[NR_TASKS];
                while (--i) {
                        if (!*--p)
                                continue;
                        if ((*p)->state == TASK_RUNNING \&\& (*p)->counter > c)
                                c = (*p)->counter, next = i;
                }
                if (c) break;
                for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                        if (*p)
                                (*p)->counter = ((*p)->counter >> 1) +
                                                (*p)->priority;
        }
        switch_to(next);
}
```

Listing 6.2: Incitatus, ProcessManager.c

```
PUBLIC void ProcessManager_switch(Regs* context) {

    Process* currentProcess = Scheduler_getCurrentProcess();
    Debug_assert(currentProcess != NULL);

    if(currentProcess == kernelProcess) { /* Switching from kernel process */

        currentProcess->kernelStack = context; /* Save process state */
        currentProcess->status = PROCESS_WAITING;

    } else { /* Switching from a user process */

        if(currentProcess->status == PROCESS_TERMINATED) { /* Switching from a *killed* user process, free resources */

            ProcessManager_destroyProcess(currentProcess);

        } else {

            if(Scheduler_isPreemptive()) {

                if(currentProcess->status != PROCESS_BLOCKED)
                    currentProcess->status = PROCESS_WAITING;

            } else {

                currentProcess->status = PROCESS_WAITING;

            }

            currentProcess->userStack = context; /* Save process state */

        }

    }

    /* Get next process from scheduler */
    Process* next = Scheduler_getNextProcess();
    Debug_assert(next != NULL);

    while(next->status == PROCESS_BLOCKED) /* Find a waiting process */
        next = Scheduler_getNextProcess();

    Debug_assert(next->status == PROCESS_WAITING);
    next->status = PROCESS_RUNNING;
    if(currentProcess == next) /* No need for a context switch */
        return;

    Debug_assert(next->kernelStack != NULL);

    if(next == kernelProcess) { /* Next process is kernel process */

        asm volatile("mov %0, %%DR0" : : "r" (next->kernelStack)); /* Store kernel process' ESP in DR0 register */

    } else { /* Next process is a user process */

        Debug_assert(next->userStack != NULL);

        GDT_setTSS(KERNEL_DATA_SEGMENT, (u32int) next->kernelStack);
        asm volatile("mov %0, %%DR0" : : "r" (next->userStack)); /* Store next process' ESP in DR0 register */

    }

    VirtualMemory_switchPageDir(next->pageDir); /* Switch to next process' address space */

}
```

6.1.3 Extendibility

Extendibility was achieved through the use of kernel module interfaces and a minimal C library. Kernel module interfaces allow students to extend the functionality of the operating system by providing their own implementations for operating system components. On the other hand, the C library lets students produce applications for the user space. Chapter 5 explains how to modify and extend the operating system in more detail.

6.2 Evaluation

6.2.1 Fulfilment of Objectives

As presented in the Introduction chapter, the project had the following objectives:

- *"Supplement operating systems classes, build up on theory by providing hands-on experience on common OS primitives."*

This objective has been satisfied. Incitatus is made up of common operating system primitives such as memory manager, process manager, scheduler and file system. Most of these components provide interfaces, giving students the ability to change the functionality of the operating system in a straightforward way.

- *"Demonstrate how modern monolithic operating systems on a modern micro-architecture operate."*

This objective has been satisfied. Incitatus is a monolithic operating system with preemptive multitasking support (through the round-robin scheduler implementation). Incitatus is developed for the x86-32bit CPU architecture which is a relatively modern architecture.

- *"Evaluate the developed operating system for its usefulness in teaching."*

This objective has only been partially satisfied. Given the modularity (modular kernel), modifiability (code readability and comments) and extensibility (module interfaces and minimal user space library) of the operating system, it has the essential attributes to be a useful instructional tool. Unfortunately, given the time constraints, it was not possible to evaluate the OS's effectiveness with a study involving student participants. Thus, this objective may further be validated in a future study that measures Incitatus' pedagogical usefulness.

6.2.2 Design

As demonstrated in the previous chapters, modularity was accomplished by individual kernel modules which represent a specific operating system primitive. These modules are statically loaded; that is, the modules are compiled and linked altogether. Thus, modularity in Incitatus is achieved through careful source code organisation and encapsulation. This approach only provides source-code modularity, which may seem relatively adequate towards accomplishing project's objectives; but the truth is, it is far from perfect.

If I had my current experience and knowledge when I first began designing Incitatus, I would have opted for dynamically loaded modules. Dynamically loadable modules are loaded in runtime and results in a truly decoupled design. Other benefits include tighter security and potentially independent build cycle for every module.

Often, the most critical design decision concerning operating systems is the kernel architecture, which certainly holds for Incitatus. The very first design decision was deciding whether Incitatus would have a monolithic or a microkernel architecture. At the time I found monolithic kernels easier to understand and implement compared to microkernels (I figured most of the other CS students also would share the same experience). Thus, Incitatus' kernel was decided to be monolithic. As the code base grew, the more unreliable the operating system became; soon after, tracking the bugs and fixing them became a hide and seek contest. If I had more experience while choosing the kernel architecture, I probably would have leant towards a microkernel design instead of a monolithic one.

6.2.3 Implementation

The implementation was by far the longest phase; as I had to digest both operating system concepts and the x86 architecture simultaneously while also developing the operating system. At the end, I was fairly pleased with the code quality (in terms of readability and modularity).

However, the latest version of the operating system suffers from a range of critical and not-so-critical software bugs; which are mainly memory leaks or stack corruptions. I have pinpointed the memory leak (non-critical) to process creation/destruction and the stack corruption (critical, results in a triple fault and a reboot) happens infrequently if the scheduler implementation is FCFS(first come, first served). Because of time constraints, I was not able to fix them. Thus, I was left dissatisfied since such software bugs simply should not exist in an operating system (especially the kernel).

6.2.4 Testing

Software testing was probably the weakest software engineering process. Largely because of the fact that automated testing wasn't utilised; instead all the tests were carried out manually. After developing a certain portion of the operating system, the new functionality (or modification) was merely tested "in-line". That is, no dedicated test files or functions were created.

Creating automated tests are not so straightforward while dealing with the hardware, as it is necessary to carry out the tests while also running the software on a deployed machine (unlike a typical build cycle such as: *Compile* $\rightarrow$ *Test* $\rightarrow$ *Deploy*). Although, later in the development process, I figured out that it is indeed possible to make use of automated tests by a specialised test driver inside the kernel. Unfortunately, given the time constraints, I wasn't be able to implement such a driver.

Chapter 7 - Conclusion

This chapter concludes the dissertation by summarising the effectiveness of the proposed solution and giving an overview of the experience gained throughout the project. Lastly, we consider further development areas based on this project.

7.1 Project Aim

The primary aim of this project is to ease the introduction of operating system concepts by developing a modular, modifiable and an extendible operating system. To accomplish this aim, an instructional operating system, Incitatus was produced. The produced operating system allows students to experiment with various operating systems concepts; potentially improving their understanding through hands-on experience.

Modularity was achieved through careful division of functionality across OS primitives; every OS primitive is a module entity. Extendibility and modifiability was achieved through kernel module interfaces, a relatively easy-to-read code base and an ability to create applications for the user-space through a user-space software library.

Considering the modularity, modifiability and the extendibility of the operating system, this project would be of use to students in understanding OS concepts. However this conclusion could be made stronger by a further examination involving undergraduate computer science students.

7.2 Personal Development

As stated in the Introduction chapter, it was the desire to explore the internal workings of operating systems that kindled this project. Throughout the development of Incitatus, I have gained invaluable insight on how operating systems operate. I also began to appreciate the conveniences that are provided by modern operating systems that we often take as granted.

Building an operating system requires a solid understanding of many computer science fields such as data structures, algorithms, concurrency, software design, hardware architecture, security and reliability. After developing Incitatus, I feel that my knowledge in said areas have been positively affected.

Operating system concepts learned during the implementation of Incitatus include:

- The definition of "Operating System"
- Kernel Architectures
- Resource Management
- Scheduling
- Processes and Concurrency

Operating systems development is an extremely practical subject, thus we can not disregard the importance of practical skills gained throughout the project. Before venturing on this project, I had almost non-existent hardware knowledge and even less so in regards to low-level systems programming. I now feel much more confident in said areas, particularly surrounding the intel x86 architecture family.

7.3 Future Work

Operating systems can never be considered as final as long as the hardware and software technologies, that drive their development, keep improving. Thus, an enormous number of improvements could be listed. Nevertheless, only the most crucial ones are considered in order to keep them manageable.

There are some rough areas in the produced operating system that could have been done better. First and foremost: the current file system implementation is unfortunately read-only; thus, the operating system would greatly benefit from a better file system implementation with hard disk support.

It would be beneficial to port the operating system to other CPU architectures such as ARM or MIPS; which are relatively simpler than the x86 architecture in complexity as they are both RISC (reduced instruction set computer) architectures as opposed to CISC (complex instruction set computer).

Incitatus lacks a proper POSIX interface. A conforming POSIX interface implementation would greatly enhance software compatibility with other UNIX-like operating systems.

As is, the project makes use of extensive code comments to aid understanding and ease modifiability. A further step would be to design exercises that let students implement a certain operating system primitive. For example, other instructional operating system projects (the current work which was presented in 2.2) all provide practical exercises on implementing certain portions of the operating system. This would also open up the possibility of a follow-up project that investigates the pedagogical effectiveness of the operating system by introducing such carefully designed exercises.

7.4 Concluding Remarks

While developing the Incitatus operating system, I was also a test subject; measuring the effectiveness this project has on undergraduate computer science students. Given the knowledge I have gained as a "test subject", it is my sincere hope that this project would be of value to students in understanding how operating systems work.

References

[1] Gnu binutils. `http://www.gnu.org/software/binutils/`. Accessed: 2013-4-07. 12

[2] Bochs emulator. `http://bochs.sourceforge.net/`. Accessed: 2013-4-07. 12

[3] Gnu compiler collection. `http://gcc.gnu.org/`. Accessed: 2013-4-07. 12

[4] Gdb: The gnu project debugger. `http://www.gnu.org/software/gdb/`. Accessed: 2013-4-07. 12

[5] Git vcs. `http://git-scm.com/`. Accessed: 2013-3-28. 4, 12

[6] Grub bootloader. `http://www.gnu.org/software/grub/`. Accessed: 2013-4-07. 13, 24

[7] Netwide assembler. `http://www.nasm.us/`. Accessed: 2013-4-07. 12

[8] Qemu emulator. `http://wiki.qemu.org/Main_Page`. Accessed: 2013-4-07. 12

[9] musl libc. `http://www.musl-libc.org/`. Accessed: 2013-4-25. 38

[10] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. Third edition, 2005. 16, 19

[11] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. *The Nachos Instructional Operating System*. URL `http://www.cs.washington.edu/homes/tom/nachos/nachos.ps`. 10, 11

[12] Free Software Foundation. Multiboot specification. `http://www.gnu.org/software/grub/manual/multiboot/multiboot.html`. Accessed: 2013-4-15. 24

[13] Randall Hyde. *The Art of Assembly Language.* Second edition, 2010. 11

[14] The IEEE and The Open Group. Posix. `http://pubs.opengroup.org/onlinepubs/9699919799/`. Accessed: 2013-4-21. 38

[15] Intel. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 1.* . URL `http://download.intel.com/products/processor/manual/253665.pdf`. 16, 17

[16] Intel. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3.* . URL `http://download.intel.com/products/processor/manual/325384.pdf`. 18, 28

[17] Doug Lea. Doug lea's malloc. `http://g.oswego.edu/dl/html/malloc.html`. Accessed: 2013-4-11. 20, 22

[18] Jonathan Levin. *Mac OS X and iOS Internals: To the Apple's Core.* First edition, 2012. 16

[19] Ben Pfaff, Anthony Romano, and Godmar Back. *The Pintos Instructional Operating System Kernel.* URL `http://benpfaff.org/papers/pintos.pdf`. 11

[20] Dennis M. Ritchie. The development of the c language. `http://cm.bell-labs.com/cm/cs/who/dmr/chist.html`. Accessed: 2013-4-07. 12

[21] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, 1987. 10

[22] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, third edition, 2007. 6, 7, 10

[23] Ken Thompson. *The Tanenbaum-Torvalds Debate.* 1992. URL `http://oreilly.com/catalog/opensources/book/appa.html`. Accessed: 2013-4-01. 6

[24] Wikipedia. Minix. `http://en.wikipedia.org/wiki/MINIX`, . Accessed: 2013-4-05. 10

[25] Wikipedia. List of operating systems. `http://en.wikipedia.org/wiki/List_of_operating_systems`, . Accessed: 2013-3-28. 5

[26] Wikipedia. tar. `http://en.wikipedia.org/wiki/Tar_(computing)`, . Accessed: 2013-4-14. 23

[27] Wikipedia. Paging. http://en.wikipedia.org/wiki/Page_table, . Accessed: 2013-4-25. vi, 31

[28] Wikipedia. Architecture of windows nt. http://en.wikipedia.org/wiki/Architecture_of_Windows_NT, . Accessed: 2013-4-16. 16

[29] Windows. Flat memory model. http://technet.microsoft.com/en-us/library/cc767886.aspx. Accessed: 2013-4-11. 19