EPQ Product Written Report: Building a 32-bit Operating System

12th December 2024

Abstract

An operating system (OS) is a collection of programs that provide basic, low-level features required for a computer. An OS has several key features (Watson & Williams 2021):

- Providing a hardware-agnostic application interface
- Managing resources and processes
- Providing a basic user interface
- Enforcing process security and isolation

OSes communicate with hardware, software, and the user and manage computer resources to effectively perform these tasks(see fig 1). Due to their requirements, Operating systems are extremely complex, and have grown in size over time to support increasingly complex hardware . For example, the Linux Kernel has grown from around 10243 LoC (lines of code) to $\geq$ 37 million LoC in version 6.11-rc4 (counted by me using `cloc`). The size can make OS development daunting to new systems programmers.

In this project, I aim to develop a basic operating system subset for IBM PC-like designs supporting modern processor features like Protected mode, virtual memory, paging, and modern executable interfaces. In doing this, I aim to improve my own skills in programming without abstraction and my understanding of Data Structures & Algorithms.

1 Introduction

An operating system (OS) is a collection of programs that provide basic, low-level features required for a computer. An OS has several key features (Watson & Williams 2021):

- Providing a hardware-agnostic application interface
- Managing resources and processes
- Providing a basic user interface
- Enforcing process security and isolation

OSes communicate with hardware, software, and the user and manage computer resources to effectively perform these tasks(see fig 1).

Due to their requirements, Operating systems are extremely complex, and have grown in size over time to support increasingly complex hardware . For example, the Linux Kernel has grown from around 10243 LoC (lines of code) to $\geq$ 37 million LoC in version 6.11-rc4 (counted by me using `cloc`). The size can make OS development daunting to new systems programmers.

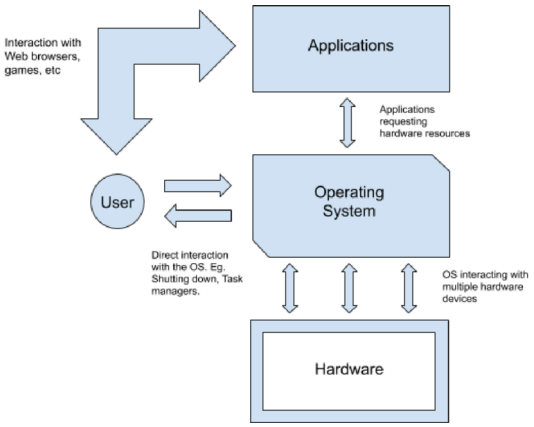


Figure 1: High-level block diagram of an operating system

My interest in operating systems comes primarily from my own experience in configuring them and tinkering with the internals of modern releases of Linux and FreeBSD, both common operating systems. Changing settings and tracking the impact on performance got me interested in OS development. I aimed to write a simple operating system which used a subset of the features I was tweaking on my own system.

2 Aims

1. To write a basic OS subset targeted towards IBM PC-like designs.
2. To build my skills in programming without abstraction (close to hardware)
3. To improve my Data Structures & Algorithms skills

3 Research

3.1 Initial Reading

The IGCSE textbook taught me the basics of the Von Neumann model, which is the computing model applied in almost all modern processors, including the `i386` (Watson & Williams 2021).

Other resources that proved vital as I was beginning my research included the OSDev Wiki (*Expanded Main Page - OSDev Wiki* n.d.). The authority and accuracy of this source were occasionally dubious; however, the "Further Reading" sections and citations were excellent for gathering further sources.

3.2 Further Reading

I found two excellent sources of information on the `i386` processor. The first of these two is Ralf Brown's Interrupt List (Brown 2000), which exhaustively detailed the IBM PC's behaviour. This, in conjunction with Intel's official datasheets for the `i386` (Intel Corporation & Intel Corporation Staff 1987), made up most of my further reading about the `i386`.

Where the `i386` reference was cumbersome e.g. when looking up single instructions / condition codes, the x86 and amd64 Instruction Reference (Cloutier 2023) was invaluable. It is a script-separated version of the Intel official reference, and was an enormous time-saver when checking simple condition codes.

For other peripherial chips, I used official datasheets from their respective suppliers, like the PS/2 datasheet from Altium (Altium Limited 2009), and the the `i8259` datasheet available from Intel (Corporation 1988).

3.3 Primary Research - Existing Artefact Analysis

I chose to include artefact analysis after finding valuable pointers in source code for my project. One of the sources I looked over was MikeOS (Saunders et al. 2023), a simple real-mode (16-bit) operating system.

The other operating system I looked at was Linux; the codebase often contained helpful pointers to the bugs in my own code in the form of comments. Linux is the world's most popular operating system today, so looking at its `0.01` release was a valuable source of information (Torvalds 1991). Though the comments contained profane language, this did not significantly impact the credibility of the source as it is the world's most popular operating system (StatCounter 2023).

Overall, my primary research was useful as it included real code that provided me with valuable pointers (but not solutions) to the problems I was facing.

3.4 Source Testing

I made sure to test the quality of every source by evaluating its currency, relevance, authorship, accuracy, and purpose. I have included a few examples of these tests below:

3.4.1 `i386` Reference

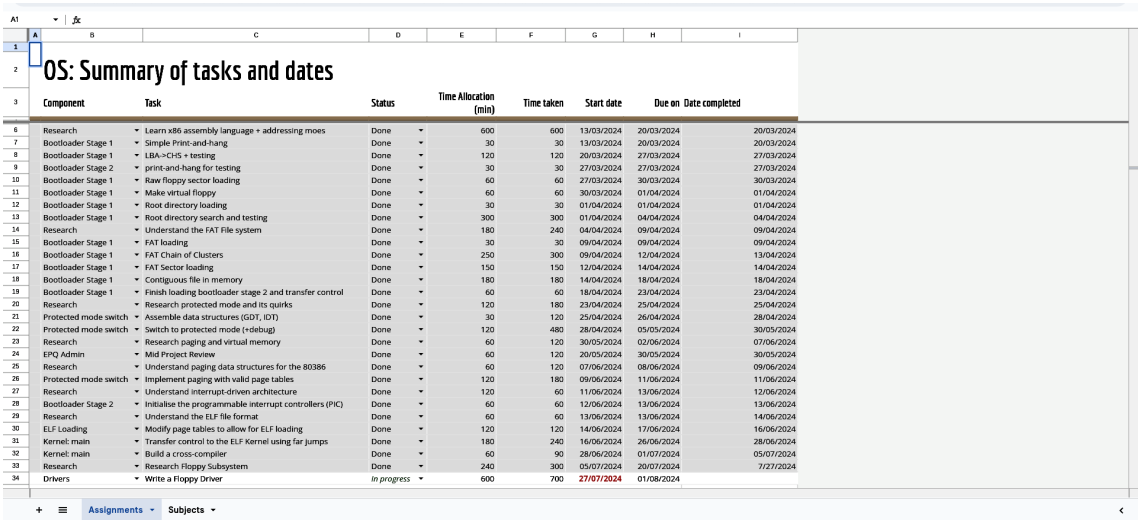
Criterion	Evaluation	Score
Currency	The source was published in 1987 - not very recent.	4/10
Relevance	The source is expressly for systems/application developers - extremely relevant.	9/10
Accuracy	The source is extremely accurate, with few known errata.	9/10
Authority	The source is published by the manufacturers/designers of the chip. The source is an official document. The author is trustworthy.	9/10
Purpose	To introduce programmers to the `i386`, which was new at the time.	9/10

This source scores 40/50 on the CRAAP test, and is extremely reliable. This was an example of an extremely high-quality source.

3.4.2 OSDev Wiki

Criterion	Evaluation	Score
Currency	The source is updated frequently; very recent.	8/10
Relevance	The source is expressly for operating system development - extremely relevant.	9/10
Accuracy	The source had a few errors.	4/10
Authority	The source is published by anonymous authors.	3/10
Purpose	To support developers in writing Operating systems.	8/10

This source scores 32/50 on the CRAAP test. It is an average quality source, but still useful.



OS: Summary of tasks and dates

Component	Task	Status	Time Allocation (min)	Time taken	Start date	Due on	Date completed
Research	Learn x86 assembly language + addressing moes	Done	600	600	13/03/2024	20/03/2024	20/03/2024
Bootloader Stage 1	Simple Print-and-hang	Done	30	30	13/03/2024	20/03/2024	20/03/2024
Bootloader Stage 1	LBA->CHS + testing	Done	120	120	20/03/2024	27/03/2024	27/03/2024
Bootloader Stage 2	print-and-hang for testing	Done	30	30	27/03/2024	27/03/2024	27/03/2024
Bootloader Stage 1	Raw floppy sector loading	Done	60	60	27/03/2024	30/03/2024	30/03/2024
Bootloader Stage 1	Make virtual floppy	Done	60	60	30/03/2024	01/04/2024	01/04/2024
Bootloader Stage 1	Root directory loading	Done	30	30	01/04/2024	01/04/2024	01/04/2024
Bootloader Stage 1	Root directory search and testing	Done	300	300	01/04/2024	04/04/2024	04/04/2024
Research	Understand the FAT File system	Done	180	240	04/04/2024	09/04/2024	09/04/2024
Bootloader Stage 1	FAT loading	Done	30	30	09/04/2024	09/04/2024	09/04/2024
Bootloader Stage 1	FAT Chain of Clusters	Done	250	300	09/04/2024	12/04/2024	13/04/2024
Bootloader Stage 1	FAT Sector loading	Done	150	150	12/04/2024	14/04/2024	14/04/2024
Bootloader Stage 1	Contiguous file in memory	Done	180	180	14/04/2024	18/04/2024	18/04/2024
Bootloader Stage 1	Finish loading bootloader stage 2 and transfer control	Done	60	60	18/04/2024	23/04/2024	23/04/2024
Research	Research protected mode and its quirks	Done	120	180	23/04/2024	25/04/2024	25/04/2024
Protected mode switch	Assemble data structures (GDT, IDT)	Done	30	120	25/04/2024	26/04/2024	28/04/2024
Protected mode switch	Switch to protected mode (+debug)	Done	120	480	28/04/2024	05/05/2024	30/05/2024
Research	Research paging and virtual memory	Done	60	120	30/05/2024	02/06/2024	07/06/2024
EPQ Admin	Mid Project Review	Done	60	120	20/05/2024	30/05/2024	30/05/2024
Research	Understand paging data structures for the 80386	Done	60	120	07/06/2024	08/06/2024	09/06/2024
Protected mode switch	Implement paging with valid page tables	Done	120	180	09/06/2024	11/06/2024	11/06/2024
Research	Understand interrupt-driven architecture	Done	120	60	11/06/2024	13/06/2024	12/06/2024
Bootloader Stage 2	Initialise the programmable interrupt controllers (PIC)	Done	60	60	12/06/2024	13/06/2024	13/06/2024
Research	Understand the ELF file format	Done	60	60	13/06/2024	13/06/2024	14/06/2024
ELF Loading	Modify page tables to allow for ELF loading	Done	120	120	14/06/2024	17/06/2024	16/06/2024
Kernel: main	Transfer control to the ELF Kernel using far jumps	Done	180	240	16/06/2024	26/06/2024	28/06/2024
Kernel: main	Build a cross-compiler	Done	60	90	28/06/2024	01/07/2024	05/07/2024
Research	Research Floppy Subsystem	Done	240	300	05/07/2024	20/07/2024	7/27/2024
Drivers	Write a Floppy Driver	*In progress*	600	700	27/07/2024	01/08/2024	

Assignments | Subjects

Figure 3: Planning method post-MPR - visualisation in Fig. 4

4 Planning

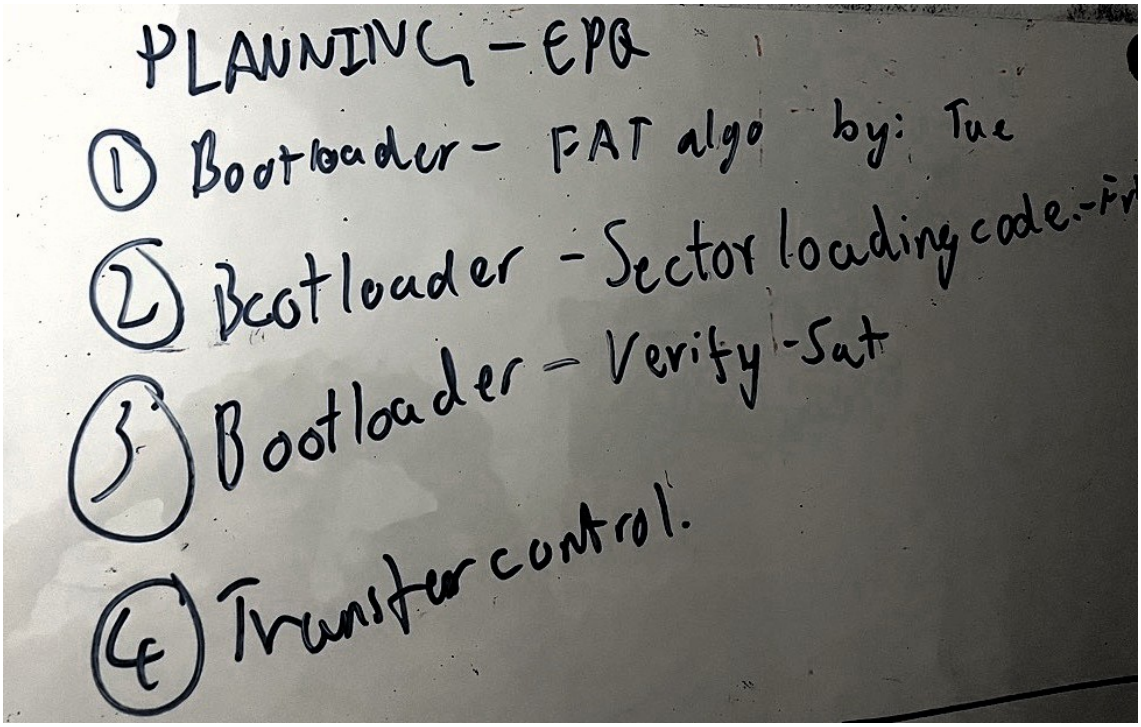


Figure 2: My initial planning method

Pre-MPR, my planning method was slightly rudimentary. I detailed a high-level plan in the production log, and then split the tasks up into several sub-tasks of manageable size (around 1 day's worth of work). I then wrote 6-7 subtasks per week on a whiteboard and then struck them off as I achieved the milestones. I then erased the whiteboard at the end of the week and wrote the next few subtasks.

Figure 4: A Gantt chart of my plans (the blank spaces are for AS-levels and a separate research project)

This planning method was not particularly effective for the EPQ as there was no record of my progress. Post-MPR, however, I switched to using a spreadsheet to track my progress. This had several advantages, as I could categorize my work into different classes, record dates when work was completed, and keep track of how closely I was adhering to my plan.

5 Operating System Development

5.1 Booting

5.1.1 1st stage bootloader

```
11 BITS 16
12 ORG 0x7C00
13 JMP BEGIN
14 NOP
15 BPB:
16 ; written with help from the good folks at OSDev and Wikipedia
   ↪  (thanks btw) and random documentation I found online
17 ; Mountable on Linux; maybe not so on MS-DOS (Windows)
18 OEMLBL: DB "VOSFLP  "
19 BYTESPERSECTOR: DW 512
20 SECTORSPERCLUSTER: DB 1
21 RESERVEDSECTORS: DW 1                ; that's us!
22 NUMBEROFFATS: DB 2                  ; as is standard on all FAT12
   ↪  systems
23 ROOTDIRENTRIES: DW 224                ; total theoretical root dir
   ↪  entries.
24 SECTORS: DW 2880                   ; Total sectors
25 MEDIADESCRIPTOR: DB 0xF0           ; am floppy
26 SECTORSPERFAT: DW 9
27 ; end DOS 2.0 BPB
28 ; begin DOS 3.31 BPB
29 SECTORSPERTRACK: DW 18
30 HEADS: DW 2
31 HIDDENSECTORS: DD 0
32 LARGESECTORS: DD 0
33 DRIVENO: DB 0                        ; am floppy!!
34 EXTBOOTSIGNATURE: DB 0x29          ; AM FLOPPY!!1!!!1!
35 SERIAL: DD 0xACDC                  ; yes
36 LABEL: DB "VOSFLOPPY  "                ; volume label ALWAYS 11
   ↪  CHARS
37 FILESYSTEM: DB "FAT12   "
```

Listing 1: The standardised data structures in the MBR, required to read or write from the disk

Bootloaders take the form of 512 raw bytes written to the first 512 bytes on the disk, a region termed the Master Boot Record. This section of the disk has a specified format.

After this table is the beginning of the bootloader, whose task it is to read the floppy and load the rest of the operating system into memory. Given that the above table takes up around 40 bytes, only 480 bytes remain for the bootloader, which

means the bootloader must be small. Since a large quantity of hardware must be initialised to use protected mode. I used a second-stage bootloader to initialise the processor and peripherals.

5.1.2 2nd stage bootloader

I used this stage to initialise only the CPU and prepare it to run the kernel, which was entirely 32-bit in nature. This meant I simply loaded the kernel into memory, and then started fully initialising the processor's different modes, continuing the booting process.

```
338         mov esi, 0x40000
339         push 0x8                    ; code segment, DPL=0
340         push dword [esi+e_entry]; e_entry
341         retf                          ; far return (far calls triple
            ↪  fault for whatever reason)
342
```

Listing 2: The 2nd stage bootloader transferring control to the kernel at the end of its execution

I used the 2nd stage bootloader to switch the processor from operating in 16-bit real mode (its initial state) to 32-bit protected mode. I then initialised memory by writing page tables, and initialised paging (a memory-managing technique). I also initialised the interrupts subsystem, which allows the processor to handle peripheral communications, loaded and interpreted the kernel, and transferred control over to it.

5.2 Kernel

```
35 int main(void) {
36         /* TO ADD - set up GDT with TSS */
37         char * hello = "hello, ELF World!\n";                    /* test
           ↪  string */
38         vga_puts(hello,VGA_ATTRIB(VGA_BLINK | VGA_BLACK, VGA_BRIGHT |
           ↪  VGA_WHITE));
39         init_interrupts(idt);                                    /* fill
           ↪  with default handler */
40
41         cli();
42         init_8259();                                            /*
           ↪  initialise the PIC */
43         lidt(idt);
44         init_8042();                                            /*
           ↪  initialise the keyboard controller */
45         sti();                                                   /*
           ↪  enable interrupts */
46         init_82077a();                                           /*
           ↪  initialise the floppy controller */
47         void * sect = read_sectors(1,1,0);
48         for(;;)
49                 asm volatile
                   ↪  ("hlt");                                    /* stop
                   ↪  here and wait for interrupts */
50 }
```

Listing 3: The main kernel file that receives control from the 2nd stage bootloader

The kernel is the core of the operating system, and manages memory, disk access, processes, CPU time, and peripherals. Developing a kernel is key to operating system development, but it is also the portion that takes the most technical skill, as correctly interfacing with peripherals is difficult. The initial pre- Mid Project Review scope of my EPQ involved developing a full kernel, with process management, but I reduced the scope of my project to include only hardware and memory management due to difficulties with implementation.

5.2.1 Interrupts

Interrupts from hardware / peripherals are handled by two chips termed "Programmable Interrupt Controllers". These chips are fairly easy to program, and once set up along with the corresponding memory structures, pass on hardware interrupts and exceptions to the operating system. These chips were set up alongside the interrupt memory structures in the 2nd stage bootloader, and required only installation of interrupt handlers that were part of the kernel.

```
37
38 __attribute__((interrupt)) void
   ↪  generic_interrupt_handler(isr_savedregs * u)
39 {
40         /* this works but is an annoyance at best. Replace with
           ↪  device
41          * driver ISR at the soonest
42          */
43         /* __asm__ volatile ("hlt"); */
44         vga_puts("Interrupt\n",0x4f);
45         return;
46 }
```

Listing 4: The default interrupt handler, which is replaced by other interrupt handlers

5.2.2 Keyboard Driver

Developing the keyboard driver was relatively painless. Keyboards send scancodes to their host computer along with an interrupt indicating a scancode is available to be read, which provide information about which key is pressed when. The keyboard driver handles the interrupt and gets the scancodes, interprets the scancodes according to the keymap in force, and passes on the characters and key combinations to the operating system.

Listing 5: The Keyboard interrupt handler

One problem I faced was not fully understanding the documentation detailing the chip's behaviour. An interrupt is raised every time a byte needs to be passed, which means that multiple interrupts can be raised for the same key-down or key-up event. I wrongly understood the peripheral as sending only one interrupt per key event, but using the debugger for some time helped me find this error and quickly rectify it.

5.2.3 VGA

The VGA driver was also thankfully very simple. In the IBM PC, the VGA framebuffer (a record of every pixel on the display) is memory-mapped to the physical address `0xB8000`, which means that writing to this location was in effect equivalent to printing on the screen (International Business Machines Corporation 1981). A simple memory copy sufficed for these purposes.

Listing 6: VGA printing routines

5.2.4 Floppy

The floppy driver caused me by far the greatest grief. The 82077A is an extremely complicated chip with a lot of different operating modes, parameters, etc along with

a significant list of behaviours deviating from both the specification and information available online (Corporation 1994). This chip was supposed to be used with a DMA (Direct Memory Access) controller, which would essentially allow the CPU to run other tasks while disk I/O occurred in the background. However, the setup of these chips, especially in protected and paged mode, was exceptionally complicated. This driver took up far more time than I was expecting, and it was one of the major reasons why I revised my MPR plan significantly.

Listing 7: Part of the floppy driver containing data structures necessary

6 Critical Evaluation

This task was an exceptionally challenging one for me, and it involved learning a lot of skills, from low-level programming to comprehension of technical references and datasheets. In those ways, I achieved my secondary aims. My primary aims were not realised to their full extent, however I have written a significant portion of the operating system, and not much more work is necessary to turn it into a full-fledged system. If more time was available to write, I would add in process management as part of the code. The ELF loader is already present, so an executable format is available; logic to parse the filesystem is already written; and all the necessary drivers have been completed, with the exception of the timer (which is a simple chip and will not take much time).

Bibliography

Altium Limited (2009), 'PS2 - PS/2 Controller'.
URL: *https://valhalla.altium.com/Learning-Guides/PS2-PS2_Controller.pdf*

Brown, R. e. a. (2000), 'Ralf Brown's Interrupt List'.
URL: *https://www.cs.cmu.edu/ ralf/files.html*

Cloutier, F. (2023), 'x86 and amd64 Instruction Reference'.
URL: *https://www.felixcloutier.com/x86*

Corporation, I. (1988), 'Datasheet: 8259A Programmable Interrupt Controller (8259A 8259A-2)'.
URL: *https://pdos.csail.mit.edu/6.828/2010/readings/hardware/8259A.pdf*

Corporation, I. (1994), '82077aa datasheet'.
URL: *https://www.alldatasheet.com/datasheet-pdf/pdf/167793/INTEL/82077AA.html*

Expanded Main Page - OSDev Wiki (n.d.).
URL: *https://wiki.osdev.org*

Intel Corporation & Intel Corporation Staff (1987), *386 Programmer's Reference Manual*, Intel Corporation (CA).

International Business Machines Corporation (1981), 'International Business Machines Personal Computer Technical Reference'.

Saunders, M., Dehling, E., Seyler, I., Endler, J., Beck, J., Takachuk, J., Horvat, M., van Tellingen, M., Gonta, M., Nemeth, P., Valongo, P., Sasano, T., Gorol, T., Clingman, T., Nagel, W., Saiko, Y. & Dietl, P. (2023), 'MikeOS 4.5 - Simple and educational Operating System written by Mike Saunders'.
URL: *https://www.github.com/mig-hub/mikeos*

StatCounter (2023), 'Operating System Market Share Worldwide'.
URL: *https://gs.statcounter.com/os-market-share*

Torvalds, L. (1991), 'Linux 0.01 source'.
URL: *ftp://ftp.kernel.org/pub/linux/kernel/Historic/linux-0.01.tar.gz*

Watson, D. & Williams, H. (2021), *Cambridge IGCSE and O Level Computer Science*, Hodder Education.