

IKT218 - ADVANCED OPERATING SYSTEMS

Course report

Spring 2023

Github repository:

<https://github.com/Siverteh/ikt218-osdev>

Sivert Espeland Husebø

December 31, 2023

Mandatory Self Declaration

Each student is solely responsible for familiarizing themselves with the legal aids, guidelines for their use, and rules regarding source usage. The declaration aims to raise awareness among students of their responsibilities and the consequences of cheating. Lack of declaration does not exempt students from their responsibilities.

1.	I hereby declare that my submission is my own work and that I have not used other sources or received any help other than what is mentioned in the submission.	Yes
2.	I further declare that this submission: <ul style="list-style-type: none">• Has not been used for any other examination at another department/university/college domestically or abroad.• Does not reference others' work without it being indicated.• Does not reference our own previous work without it being indicated.• Has all references included in the bibliography.• Is not a copy, duplicate, or transcription of others' work or submission.	Yes
3.	I am aware that violations of the above are considered to be cheating and can result in cancellation of the examination and exclusion from universities and colleges in Norway, according to the Universities and Colleges Act, sections 4-7 and 4-8 and the Examination Regulation, sections 31.	Yes
4.	I am aware that all submitted assignments may be subjected to plagiarism checks.	Yes
5.	I am aware that the University of Agder will handle all cases where there is suspicion of cheating according to the university's guidelines for handling cheating cases.	Yes
6.	I have familiarized myself with the rules and guidelines for using sources and references on the library's website.	Yes
7.	I have decided that the effort within the group is notably different and therefore wish to be evaluated individually. Ordinarily, all participants in the project are evaluated collectively.	No

Publishing Agreement

Authorization for Electronic Publication of Work The author(s) hold the copyright to the work. This means, among other things, the exclusive right to make the work available to the public (Copyright Act. §2).

Theses that are exempt from public access or confidential will not be published.

I hereby grant the University of Agder a royalty-free right to make the work available for electronic publication:	Yes
Is the work confidential?	No
Is the work exempt from public access?	No

Contents

List of Figures	iv
1 Introduction	1
2 Assignment 1 - Introduction to OSDev	2
2.1 What is the POST?	2
2.2 Assuming that the POST of the computer is successful, what happens next?	2
2.3 Which different bootloaders exist and why would you choose one over the other? Do any of the bootloaders you find have any benefits? How could you implement a bootloader manually?	3
2.4 How is the standard memory layout of an i386 boot process, assuming that we start the free space as memory address 0x1000000?	4
3 Assignment 2 - Terminal and Global Descriptor Table	6
3.1 Development environment	6
3.2 Booting	6
3.2.1 boot.asm:	6
3.2.2 Constants:	8
3.2.3 Section .multiboot	8
3.2.4 Section .bss	8
3.2.5 Section .text	9
3.2.6 Functionality	9
3.3 Global Descriptor Table	9
3.3.1 Theory	9
3.3.2 Descriptors in the Global Descriptor Table	10
3.3.3 gdt.h:	11
3.3.4 gdt.cpp:	13

3.3.5	gdt.asm:	15
3.3.6	Functionality	15
3.4	Printing	16
3.4.1	Theory	16
3.4.2	terminal_write	16
3.4.3	printf	18
3.4.4	Functionality	21
3.5	Approach and challenges	22
4	Assignment 3 - Interrupts	23
4.1	Interrupt Descriptor Table	23
4.1.1	Theory	23
4.1.2	Interrupts & exceptions	23
4.1.3	Descriptors in the Interrupt Descriptor Table	23
4.1.4	idt.h	25
4.1.5	idt.cpp	26
4.1.6	idt.asm	31
4.1.7	Functionality	31
4.2	Interrupt Service Routines & Interrupt Requests	32
4.2.1	Interrupt Service Routines	32
4.2.2	Interrupt Requests	33
4.2.3	isr.h	33
4.2.4	isr.cpp	35
4.2.5	interrupt.asm	38
4.2.6	Functionality	41
4.3	Keyboard Logger	42
4.3.1	keyboard.h	42

4.3.2	keyboard.cpp	44
4.3.3	Functionality:	50
4.4	Approach and challenges	51
5	Assignment 4 - Memory Management, PIT, and Pull Request	52
5.1	Memory Management	52
5.1.1	Paging	52
5.1.2	memory_management.h	52
5.1.3	memory_management.c	53
5.1.4	paging.c	60
5.1.5	Functionality	62
5.2	Programmable Interval Timer	62
5.2.1	Theory	62
5.2.2	pit.h	63
5.2.3	pit.cpp	64
5.2.4	Functionality	67
5.3	Music player	68
5.4	Successful Pull Request to Main Repo	70
5.5	Approach and challenges	70
6	Conclusion	72
	Bibliography:	73
	Online sources	74
	Picture sources	76
	List of Figures	
1	Typical POST screen example [2]	2

2	Real mode memory layout of the i386 boot process [8]	5
3	Testing my boot code	9
4	Example of a descriptor in the global descriptor table [13]	11
5	How the access byte and flags looks like [13]	11
6	Memory segments being successfully updated	16
7	Printf function output	22
8	Examples of entries in the Interrupt Descriptor Table[17]	24
9	Proof of a working IDT	32
10	Example of exceptions/interrupts working in my program	42
11	Writing to my terminal using the keyboard	50
12	Example of my memory management working	62
13	Sleeping using interrupts and busy waiting	67
14	Music player in action	69
15	Successful pull request to main repo	70

1 Introduction

This report will document the designing and programming of an operating system from scratch. The goal of this project is to gain a deeper understanding of how an operating system works. This will be done by starting with a minimalistic implementation of an operating system and gradually expanding its functionalities. The report will consist of various assignments that build upon each other and expands my operating system. Each assignment will highlight the key concepts, challenges, and design decisions I have made during the implementation process. This report will serve as a guide for anyone interested in learning about the design and implementation of an operating system.

2 Assignment 1 - Introduction to OSDev

2.1 What is the POST?

POST stands for Power-On Self Test and is as the name implies a set of diagnostic tests that the computer performs on itself as it powers on [1]. This process is the first part of the boot sequence and does not need an operating system to run. The purpose of these tests is to make sure that there are not any hardware-related issues that would stop the device from booting correctly. POST only tests the hardware components of the device and does not check for any software-related issues. This process runs tests on the random access memory, storage, processor, etc. If the POST process finds any issues it will prompt the user with POST codes in the form of beeps or error messages [1].

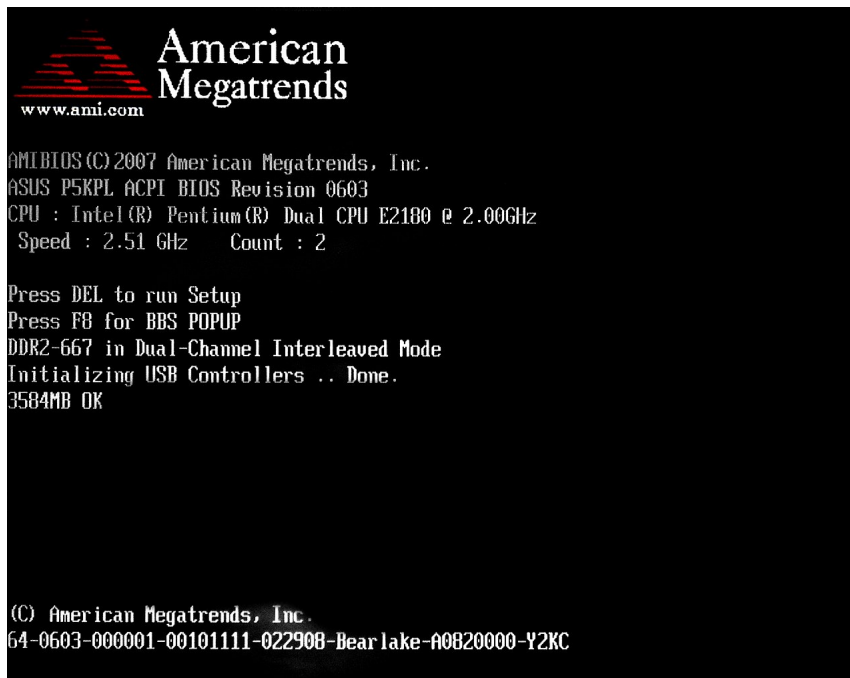


Figure 1: Typical POST screen example [2]

2.2 Assuming that the POST of the computer is successful, what happens next?

If the POST process completes successfully there are no crucial hardware-related issues on the device, or at the very least no issues that the POST tests have detected [1]. There can still appear problems after the POST process, but these problems are usually related

to the software rather than the hardware. A successful POST process means the device can continue to the next part of the boot process [1].

After the POST process, the BIOS wants to boot the operating system which is found somewhere in the storage, so the BIOS transfers control to the bootloader [3]. The bootloader then loads the kernel which is the core part of the operating system. When the kernel is loaded the operating system can start to initialize drivers, configure the system, and start-up necessary programs. Lastly, after the operating system has completed its start-up the user can interact with it through a graphical user interface or a terminal [3].

2.3 Which different bootloaders exist and why would you choose one over the other? Do any of the bootloaders you find have any benefits? How could you implement a bootloader manually?

There are several different bootloaders that exist, some of the most popular are:

- **Grub (GRand Unified Bootloader)** is one of the most popular and widely used bootloaders for Linux as well as other operating systems. It has the ability to select and boot different operating systems and kernels at startup. GRUB is a flexible and highly customizable bootloader [4].
- **Syslinux** is a lightweight bootloader designed to be used on small media devices like diskettes or USB drives. It is mainly used for booting live Linux distributions. Because of its simple configuration system, and small footprint, it is an ideal choice for systems with limited resources [4].
- **LILO (Linux Loader)** is a legacy bootloader for Linux that has been largely replaced by GRUB. It is still in use on older systems as it is a simple and reliable bootloader. LILO is rarely integrated on newer systems as it severely lacks support for modern features like disk encryption and large file systems [4].
- **Windows Boot Manager** is the bootloader used in modern Windows operating systems. It loads the Windows kernel and initializes the system components used in windows. The bootloader can be configured to boot a Linux system if this is wanted [5].

Each one of these bootloaders has its own pros and cons. Which one you choose depends on several factors like which and how many operating systems you are using, the device/hardware you are running on, the specific requirements for the system, etc. The bootloader I would say has the most benefits compared to flaws is the GRUB bootloader as it has multi-boot support, supports a wide range of file systems, has a customizable boot menu, as well as a built-in rescue mode, with its main flaw being that it can be complex to configure [4].

To implement a boot-loader manually you need to understand how the boot process works, the role of the BIOS and the basics of computer architecture [6]. Then you have to write the actual bootloader code. This code is usually written in assembly and should be optimized for size and speed as the bootloader has to fit into a very small amount of memory. Lastly, you have to create a bootable disk image that contains the bootloader code and is compatible with the system's BIOS [6].

2.4 How is the standard memory layout of an i386 boot process, assuming that we start the free space as memory address 0x1000000?

The introduction of the i386 processor was also the introduction of the first 32-bit processor as well as a new mode called protected mode [7]. Before this processors were 16-bit and ran in a mode called real mode. Real mode was used by early operating systems like Ms-DOS but came with a lot of limitations like only being able to access 1 MB of memory, no memory protection, no virtual memory, and compatibility issues. Due to these limitations, real mode was largely replaced by protected mode which offered memory protection, virtual memory, and support for 32-bit and 64-bit processing. For backward compatibility reasons, modern systems still start up in real mode before transitioning into protected mode [7].

Because the i386 processor was a 32-bit processor it could have a maximum of 4 GB of memory [8]. This memory exists and is useable but only in protected mode, the first 1MB still runs in real mode. In this first 1MB of memory we can find several reserved areas such as the Interrupt Vector Table (IVT), BIOS data area, and space for the operating systems boot loader. The memory layout in real mode for an i386 processor where the memory space starts at 0x1000000 looks like this [8]:

start	end	size	description	type	
Real mode address space (the first MiB)					
0x00000000	0x000003FF	1 KiB	Real Mode IVT (Interrupt Vector Table)	unusable in real mode	640 KiB RAM ("Low memory")
0x00000400	0x000004FF	256 bytes	BDA (BIOS data area)		
0x00000500	0x00007BFF	almost 30 KiB	Conventional memory	usable memory	
0x00007C00	0x00007DFF	512 bytes	Your OS BootSector		
0x00007E00	0x0000FFFF	480.5 KiB	Conventional memory		
0x00080000	0x0009FFFF	128 KiB	EBDA (Extended BIOS Data Area)	partially used by the EBDA	
0x000A0000	0x000BFFFF	128 KiB	Video display memory	hardware mapped	384 KiB System / Reserved ("Upper Memory")
0x000C0000	0x000C7FFF	32 KiB (typically)	Video BIOS	ROM and hardware mapped / Shadow RAM	
0x000C8000	0x000EFFFF	160 KiB (typically)	BIOS Expansions		
0x000F0000	0x000FFFFF	64 KiB	Motherboard BIOS		

Figure 2: Real mode memory layout of the i386 boot process [8]

After this first 1MB the system transitions to protected mode and the memory above 0x1000000 becomes accessible. The layout of the space beyond the first 1MB is determined by the operating systems memory management system. In general is this the place where we can find the boot code, kernel, user space, and dynamically allocated memory areas [8].

3 Assignment 2 - Terminal and Global Descriptor Table

3.1 Development environment

To start developing my operating system I am using the development environment given by Per-Arne Andersen for the course IKT218 Advanced operating systems. This development environment uses VSCode as its IDE, and already comes prefixed with a grub bootloader, makefile, cmake, and linker script [9].

The bootloader that we are using for our development environment is the grub bootloader. This bootloader loads my operating system into the computer's memory, provides the kernel with information, and transfers control to the kernel [9]. Various makefiles sets the rules and dependencies for building my program while CMake is used for my build configuration. The linker script specifies how the different files in my program should be combined and linked together [10].

3.2 Booting

It is not a specific task within assignment 2, but for me to be able to extend my operating system in any meaningful way I have to write the code necessary for booting my system. Booting an operating system is the start-up sequence that starts a computer and prepares it for use. This has several important purposes like setting up the hardware, loading the kernel, and creating a user environment.

3.2.1 boot.asm:

My implementation of the boot.asm file is based on the bare-bones tutorial by osdev [11]. This is the code used to actually boot my system.

```
1 MBALIGN equ 1 << 0
2 MEMINFO equ 1 << 1
3 MBFLAGS equ MBALIGN | MEMINFO
4 MAGIC equ 0x1BADB002
5 CHECKSUM equ -(MAGIC + MBFLAGS)
6
7 [BITS 32]
```

```

8
9  section .multiboot
10 align 4
11 multiboot:
12     dd MAGIC
13     dd MBFLAGS
14     dd CHECKSUM
15
16     dd 0
17     dd 0
18     dd 0
19     dd 0
20     dd 0
21
22     dd 0
23     dd 640
24     dd 480
25     dd 32
26
27 section .bss
28 align 16
29 stack_bottom:
30 resb 16384
31 stack_top:
32
33 section .text
34 global _start: function (_start.end - _start)
35 _start:
36     extern initialize_multiboot
37     push ebx
38     push eax
39     call initialize_multiboot
40
41     mov esp, stack_top
42
43     extern kernel_main
44     call kernel_main
45
46     cli
47 .hang:     hlt
48           jmp .hang
49 .end:

```

3.2.2 Constants:

The file starts by declaring the various constants required for the multiboot header. The multiboot header is a datastructure that is needed for the bootloader to find and recognize the operating systems kernel [11]. These constants are part of the Multiboot Standard and are "magic" numbers that are needed for a multiboot compliant bootloader to find and load the boot code. For the constants \ll means a left shift bit operation, $|$ means the bitwise or operation, and $-$ means negation. `MBALIGN` is set to $1 \ll 0$ or 1, and is used to indicate that the loaded modules must be aligned on page boundaries. `MEMINFO` is set to $1 \ll 1$ or 2, and is used to indicate that the bootloader should provide a memory map. `MBFLAGS` is set to a combination of `MBALIGN` and `MEMINFO` which will be 3, and means that the `MBALIGN` and `MEMINFO` will be enabled. The `MAGIC` constant is set to a "magic number" which is a number that lets the bootloader find the header. Lastly, the constant `CHECKSUM` is set to the negation of the sum of the `MAGIC` and `MBFLAGS` constants, and provides error-detection to verify that the Multiboot header is valid[11].

3.2.3 Section `.multiboot`

The first section of the boot code is the `.multiboot` section which defines the Multiboot header using the constants defined above. This section is where the bootloader will look for the Multiboot header and make sure the operating system is Multiboot compliant. The `align 4` is required for the Multiboot specification by making sure the Multiboot header is aligned on a 4-byte boundary. lastly, the actual Multiboot header is defined using the constants declared above [11].

3.2.4 Section `.bss`

The `.bss` section is used to declare uninitialized data. In my case, the only uninitialized data I declare is a stack of size 16Kib. This section is aligned on a 16-byte boundary for performance reasons. Lastly, I define labels for the bottom and top of the stack and reserve 16Kib of uninitialized space for the section [11].

3.2.5 Section .text

The last section in my boot code is the .text section which is where I declare the executable code of my program. This section is the starting point of my program and is where the rest of the code in the program gets run. The line `global _start:function (_start.end - _start)` declares the `_start` label as a global symbol that can be found by my linker. Its size is set to the difference between `_start.end` and `_start` labels. The external function `intitalize_multiboot` is run which is a function that checks if the multiboot check went okay or not. I also set up the stack declared in the .bss section by making the `esp` register point to the top of the stack which initializes it. The `ESP` register is also known as the current stack pointer. Lastly, i call the kernel `main` function which is the main entry point of the kernel, and runs the rest of the code in my operating system [11].

3.2.6 Functionality

To check that my boot code is working is a simple process. As long as i get into my terminal and do not get stuck on the boot video my boot code is working.



Figure 3: Testing my boot code

3.3 Global Descriptor Table

3.3.1 Theory

The x86 architecture uses the methods of segmentation and paging to provide virtual memory and memory protection [12]. A Global Descriptor Table (GDT) is the data

structure the x86 architecture uses to define and manage memory segments. This GDT contains descriptors that define the attributes of the memory segments. Each memory segment has its own descriptor with its own attributes. The GRUB bootloader already comes with a GDT built in, but for us to be able to locate and use it we have to write our own and overwrite the old one [12].

3.3.2 Descriptors in the Global Descriptor Table

Let's take a closer look at the various descriptors in the GDT. There are typically five descriptors in a GDT: the null descriptor, code descriptor, data descriptor, user mode code descriptor, and user mode data descriptor. However, only the first three of these are needed to create a working GDT. These descriptors are used to set up the segmentation registers found in the x86 architecture. Every descriptor follows the same recipe by containing the same attributes [13]. These attributes are as followed:

- **Base address:** A 32-bit value that represents the starting address of the memory block referenced by the descriptor. This 32-bit value is spread over three different areas in the descriptor: the lower 16 bits (bits 0-15), the middle 8 bits (bits 16-23), and the upper 8 bits (bits 24-31) [13].
- **Limit:** A 20-bit value that represents the size of the memory block that the descriptor references. This 20-bit value is split into two parts: the lower 16 bits (bits 0-15) and the upper 4 bits (bits 16-19) [13].
- **Access byte:** This is a group of bit flags that defines who has access to the memory referenced by the descriptor. Examples of these access flags are whether the data is readable or writeable (RW), whether or not the code is executable (Ex), the privilege level of the segment (Privl), if the segment is present in memory (Pr), and if the segment has been accessed (Ac) [13].
- **Access flags:** These influences segment size. Granularity is an access flag and determines whether the limit is specified in bytes or 4KB pages [13].

31				16		15				0			
Base 0:15						Limit 0:15							
63		56		55 52		51 48		47		40 39		32	
Base 24:31		Flags		Limit 16:19		Access Byte				Base 16:23			

Figure 4: Example of a descriptor in the global descriptor table [13]

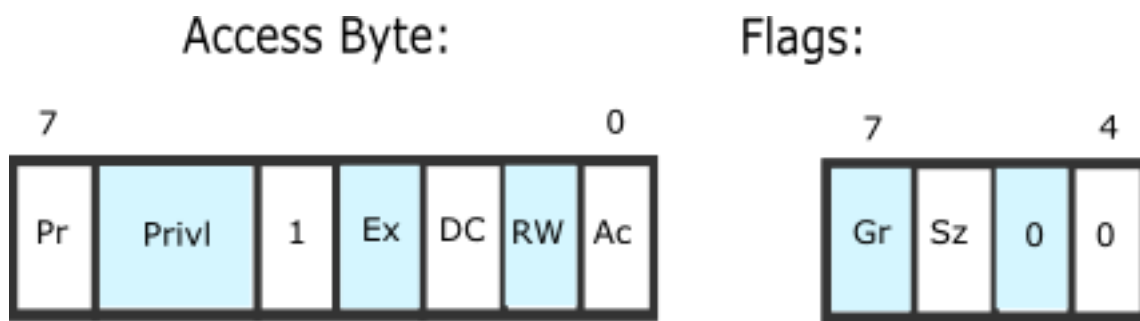


Figure 5: How the access byte and flags looks like [13]

3.3.3 gdt.h:

My implementation of the GDT is based on the operating system guide by James Molloy [12].

```

1  #include <stdint.h>
2
3  #define GDT_ENTRIES 5
4
5  struct gdt_entry_t
6  {
7      uint16_t limit_low;
8      uint16_t base_low;
9      uint8_t  base_middle;
10     uint8_t  access;
11     uint8_t  granularity;
12     uint8_t  base_high;
13 } __attribute__((packed));
14
15 struct gdt_ptr_t

```

```

16 {
17     uint16_t limit;
18     uint32_t base;
19 } __attribute__((packed));
20
21 void init_gdt();
22
23 void gdt_set_gate(int32_t num, uint32_t base, uint32_t limit, uint8_t access, uint8_t
    ↪ gran);

```

The gdt.h file starts by defining the number of entries in the GDT as 5 [12]. Then a struct called gdt_entry_t is defined. This struct represents one entry/descriptor in the GDT and contains fields representing the different components in a GDT entry. The fields in the struct are:

- **limit_low:** corresponds to the lower 16 bits of the limit of the memory segment.
- **base_low:** corresponds to the lower 16 bits of the base address of the memory segment.
- **base_middle:** corresponds to the next 8 bits of the base address of the memory segment.
- **access:** corresponds to access flags that determine which ring/privilege level the memory segment can be used in.
- **granularity:** corresponds to flags that determine the size and alignment of the memory segment.
- **base_high:** corresponds to the last 8 bits of the base address of the memory segment.
- The attribute((packed)) line ensures that the compiler does not insert any padding between the fields in the struct, which ensures the correct memory layout for the GDT.

Then another struct called gdt_ptr is defined. The purpose of this struct is to tell the processor where to find my GDT. This struct contains the size and base address of the entire GDT. The fields in the struct are:

- **limit:** corresponds to the upper 16 bits of all the selector limits.

- **base:** corresponds to the address of the first instance of the `gdt_entry_t` struct.

Lastly, the `gdt.h` file declares the functions `gdt_set_gate` and `init_gdt` [12].

3.3.4 gdt.cpp:

```

1  #include "gdt.h"
2
3  extern "C" {
4      extern void gdt_flush(uint32_t gdt_ptr);
5  }
6
7  static gdt_entry_t gdt_entries[GDT_ENTRIES];
8  static gdt_ptr_t   gdt_ptr;
9
10 void gdt_set_gate(int32_t num, uint32_t base, uint32_t limit, uint8_t access, uint8_t
    ↪ gran)
11 {
12     gdt_entries[num].base_low   = (base & 0xFFFF);
13     gdt_entries[num].base_middle = (base >> 16) & 0xFF;
14     gdt_entries[num].base_high  = (base >> 24) & 0xFF;
15
16     gdt_entries[num].limit_low   = (limit & 0xFFFF);
17     gdt_entries[num].granularity = (limit >> 16) & 0x0F;
18
19     gdt_entries[num].granularity |= gran & 0xF0;
20     gdt_entries[num].access      = access;
21 }
22
23 void init_gdt()
24 {
25     gdt_ptr.limit = sizeof(struct gdt_entry_t) * GDT_ENTRIES - 1;
26
27     gdt_ptr.base = (uint32_t)&gdt_entries;
28
29     gdt_set_gate(0, 0, 0, 0, 0);
30     gdt_set_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF);
31     gdt_set_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF);
32     gdt_set_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF);
33     gdt_set_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF);
34
35     gdt_flush((uint32_t)&gdt_ptr);
36 }

```

The `gdt.cpp` file starts by getting the external function `gdt_flush` from the `gdt.asm` file. Then I set up instances of the structs declared in my header file. I create an array of five instances of the `gdt_entry_t` struct, and one instance of the `gdt_ptr` struct.

`gdt_set_gate:`

I then set up the function `gdt_set_gate`. This function is used for setting up a descriptor for the GDT by populating the fields in the `gdt_entries` array. The function parameters are:

- **num:** a 32-bit integer that specifies the index of the descriptor in the GDT.
- **base:** a 32-bit integer that specifies the base address of the memory segment.
- **limit:** a 32-bit integer that specifies the size limit of the memory segment.
- **access:** an 8-bit integer that specifies the access flags of the memory segment.
- **gran:** an 8-bit integer that specifies the granularity of the memory segment.

The fields in the struct are set as followed:

- **base_low:** is set to the low 16 bits of the base address of the memory segment.
- **base_middle:** is set to the next 8 bits of the base address of the memory segment.
- **base_high:** is set to the high 8 bits of the base address of the memory segment.
- **limit_low:** is set to the low 16 bits of the limit of the memory segment.
- **granularity:** first sets the high 4 bits of the granularity as the high 4 bits of the limit of the memory segment. Then sets the low 4 bits of the granularity as the `gran` parameter.
- **access:** is set as the access parameter.

`init_gdt:`

Lastly, I set up the function `init_gdt`. This function sets up all the descriptors for the GDT and loads it into memory. The first line sets the limit of the `gdt_ptr` as the size of one `gdt_entry_t` struct multiplied by the number of entries minus one as its index starts from zero. Next, the base of the `gdt_ptr` is set to the address of the first entry in the

gdt_entries array. Then I set up a descriptor for each of the memory segments of the GDT. I set up descriptors for the null segment, code segment, data segment, user mode code segment, and user mode data segment are all set up using the gdt_set_gate function. Lastly, I load the GDT into the processor's memory using the function gdt_flush.

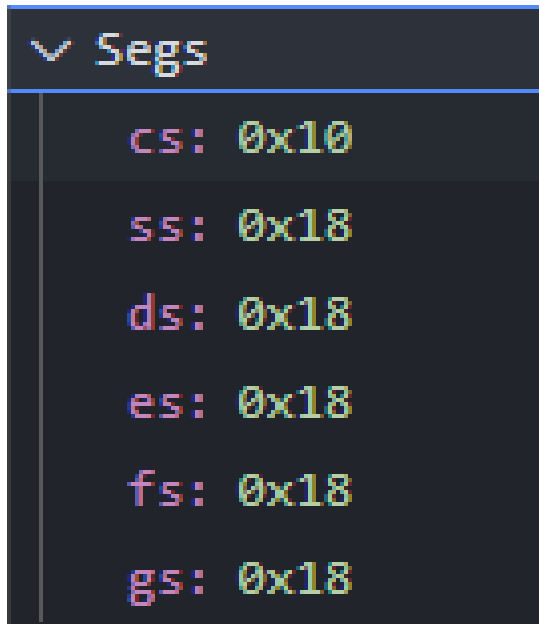
3.3.5 gdt.asm:

```
1  [GLOBAL gdt_flush]
2
3  gdt_flush:
4      mov eax, [esp+4]
5      lgdt [eax]
6
7      mov ax, 0x10
8      mov ds, ax
9      mov es, ax
10     mov fs, ax
11     mov gs, ax
12     mov ss, ax
13     jmp 0x08:.flush
14 .flush:
15     ret
```

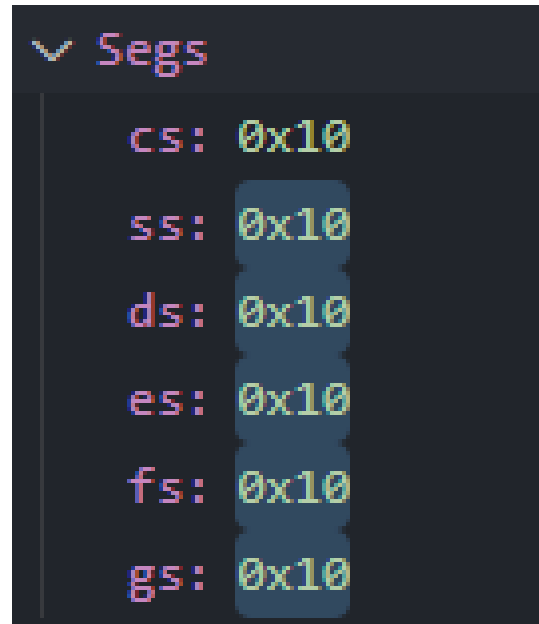
The only purpose of the gdt.asm file is to set up the function gdt_flush to be used in gdt.cpp file [12]. The file starts by defining the function as global so it can be used in other files. Then I declare the actual function gdt_flush. The first line of the function loads the address of the new GDT into the eax register which is then used to load the new GDT into the processor's Global Descriptor Table Register (GDTR) using the lgdt assembly instruction. The 0x10 selector is the offset to the data segment descriptor in my GDT. By loading all of my data segment selectors with this value I essentially activate my new GDT. Lastly, I jump to our code segment descriptor located at 0x08. This is to ensure that the processor's code segment register is properly updated [12].

3.3.6 Functionality

I can check that my Global Descriptor Table have been successfully loaded by checking that all the segment registers have been updated by the value 0x10:



(a) Old GDT



(b) New GDT

Figure 6: Memory segments being successfully updated

3.4 Printing

3.4.1 Theory

The x86 architecture uses the VGA (Video Graphics Array) text mode buffer to display text on the screen [14]. This buffer keeps track of the entire screen by being structured as a two-dimensional grid. Each row in the grid corresponds to a line of text on the screen and each column corresponds to the characters position in that line. For example if a character is added to the very start of the buffer it will be printed at the top left corner of the screen. The VGA text mode buffer is usually located at the memory address 0xB8000. Each character added to this buffer usually consists of two bytes. One of the bytes is for the actual printable ascii character while the other byte holds the characters color and attributes. This setup gives me the ability to manipulate the screen on a character-by-character basis [14].

3.4.2 terminal_write

I started my program's printing functionality by implementing a function called terminal_write. The point of this function would be to get some text to appear in the terminal

and it would not have any functionality for printing integers, floats, chars, or additional strings. Once I had the basic functionality for printing to the screen I could later expand my function with more sophisticated functionalities.

```
1 void terminal_write(char* str)
2 {
3     uint16_t* VideoMemory = (uint16_t*)0xb8000;
4
5     static uint8_t x=0, y=0;
6
7     while(*str != '\0')
8     {
9         switch(*str)
10        {
11            case '\n':
12                y++;
13                x = 0;
14                break;
15            default:
16                VideoMemory[VGA_WIDTH*y+x] = (VideoMemory[VGA_WIDTH*y+x] & 0xFF00) |
↪ *str;
17                x++;
18                break;
19        }
20
21        if(x >= VGA_WIDTH)
22        {
23            y++;
24            x = 0;
25        }
26
27        if(y >= VGA_HEIGHT)
28        {
29            for(y = 0; y < VGA_HEIGHT; y++)
30            {
31                for(x = 0; x < VGA_WIDTH; x++)
32                {
33                    VideoMemory[VGA_WIDTH*y+x] = (VideoMemory[VGA_WIDTH*y+x] &
↪ 0xFF00) | ' ';
34                }
35            }
36
37        }
38        *str++;
39    }
40 }
```

The function first declares a pointer called `VideoMemory` that points to the memory address for the VGA text-mode buffer that is located at the memory address `0xb8000`. Any character whose ASCII code corresponds to a printable character that is added to this buffer will be printed to the terminal. I also declare the variables `x` and `y` that holds the cursor's current horizontal and vertical coordinates in the terminal. Then the function loops through every character in the string and a switch statement based on the current character happens. If the current character is a newline character the horizontal cursor coordinate is set to zero and the vertical cursor coordinate is incremented. For any other character, the character is added to the VGA text-mode buffer at the current cursor coordinates which prints it to the screen. A new `uint16_t` value is created with the current character stored in the lower byte of the `uint16_t` and the color attribute `0xFF00` stored in the upper byte of the `uint16_t`. This new value is then added to the VGA text-mode buffer, and the horizontal cursor coordinate is incremented. Lastly, there are two checks for if the cursor variable reaches the width or height of the terminal.

3.4.3 printf

Now that I had a function that could print text to the terminal I wanted to expand its functionality. The main new functionality of this function would be the ability to incorporate multiple parameters into the printed string. This would work similarly to how the standard library `printf` function works. To start with I wanted the function to work with the parameter datatypes `string`, `char`, `int`, and `float`. For this function to be easier to write I rewrote the `terminal_write` function into a function called `terminal_putchar` by removing the while loop. This new function takes a single character as a parameter and prints it to the terminal.

```
1 void printf(char* str, ...)
2 {
3     va_list args;
4     va_start(args, str);
5
6     while(*str != '\0')
7     {
8
```



```

9      if ( *str != '%' )
10      {
11          terminal_putchar(*str);
12          str++;
13
14          continue;
15      }
16
17      str++;
18
19      switch (*str)
20      {
21          case '%':
22          {
23              terminal_putchar('%');
24              str++;
25              break;
26          }
27
28          case 'c':
29          {
30              terminal_putchar(va_arg(args, const char*));
31              str++;
32              break;
33          }
34
35          case 's':
36          {
37              const char* s = va_arg(args, const char*);
38              while(*s != '\0')
39              {
40                  terminal_putchar(*s);
41                  *s++;
42              }
43              str++;
44              break;
45          }
46
47          case 'd':
48          {
49              int num = va_arg(args, int);
50              char nstr[50];
51
52              int_to_string(nstr, num);
53              char* ptr = nstr;
54
55              while(*ptr != '\0')
56              {

```

```

57         terminal_putchar(*ptr);
58         *ptr++;
59     }
60     str++;
61     break;
62 }
63
64 case 'f':
65 {
66     char nstr[50];
67     float_to_string(nstr, f, 2);
68     char* ptr = nstr;
69
70     while(*ptr != '\0')
71     {
72         terminal_putchar(*ptr);
73         *ptr++;
74     }
75     str++;
76     break;
77 }
78 }
79 }
80 va_end(args);
81 }

```

The `printf` function starts by declaring a `va_list` `args` and initializes the first entry as an `str` variable. The `args` list is a list that holds every parameter in the function from left to right. It has a special `...` parameter which means there can be a variable number of parameters entered in the function, and these parameters can be any datatype. Then a while loop happens that loops through every character in the entered string. If the current character is not a `%` character the `terminal_putchar` function is called and the character is printed to the terminal. Then the current character is incremented and the `continue` statements make the loop start from the top. If the current character is a `%` character the `str` variable is incremented and the next character is put through a switch statement.

For the switch statement, the character should either be a `'%'` for a percentage character, `'c'` for char, `'s'` for a string, `'d'` for an integer, or `'f'` for a float. To have the ability to print a `'%'` character to the terminal there is a special case for when there are two `'%'` characters after each other. If the character is a `'c'` the program simply finds the character param-

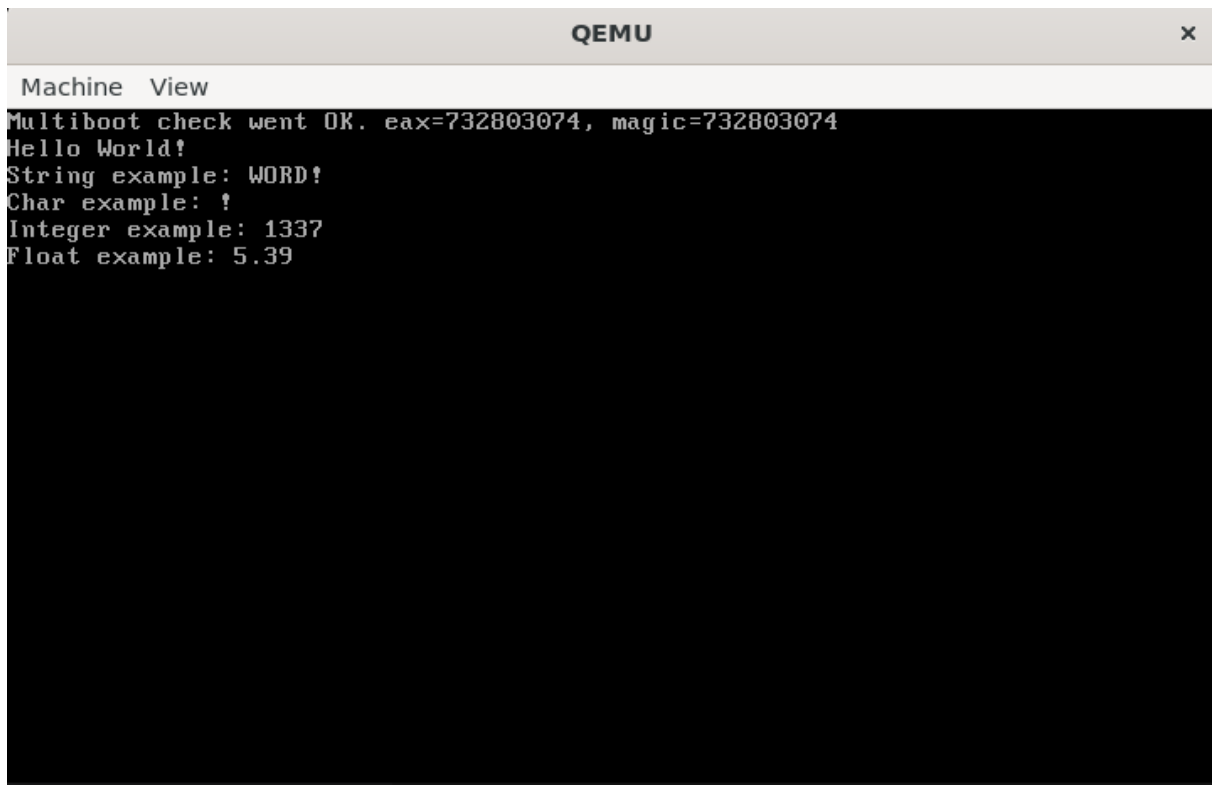
eter from the args list and prints it to the terminal using the terminal_putchar function. Similarly, if the character is an 's' it finds the string parameter from the args list loops through it, and prints every character to the terminal with the terminal_putchar function. The switch case for 'd' and 'f' works similarly. The function gets the integer or float from the args list and turns it into a string with either the int_to_string or float_to_string functions then loops through the string and prints every character to the terminal using the terminal_putchar function.

3.4.4 Functionality

To test the functionality of my printf function I used this code in main:

```
1 char* word = "WORD!";
2 char character = '!';
3 int number = 1337;
4 float float_number = 5.39;
5
6 printf("Hello World!");
7 printf("String example: %s\nChar example: %c\nInteger example: %d\nFloat example:
   ↪ %f\n", word, character, number, float_number);
```

Which gave me this output:



```
Machine View
Multiboot check went OK. eax=732803074, magic=732803074
Hello World!
String example: WORD!
Char example: !
Integer example: 1337
Float example: 5.39
```

Figure 7: Printf function output

3.5 Approach and challenges

My approach for assignment 2 was firstly to get a general feel for the development environment. I started out by just learning the basics of os development. It took me alot of playing around to figure out how to code in this environment without getting errors everywhere. Once I got a good feel for programming for an operating system I started working on the actual tasks provided in the assignment. I had alot of struggles getting my operating system to boot. None of the guides I looked at online used CMake and I thought I had to create the object files for the program myself. Once I found a guide that was appropriate for my development environment I could not get it to work as it used Bootstrap assembly instead of NASM. Eventually I figured it out and managed to boot my system. Once I had my system booted successfully the rest of assignment 2 was fairly manageable and I coded it up without a ton of problems. The main problem I faced in this phase was accidentally switching the order of the values in the gdt_entry_t struct which made me unable to load my GDT. The printing functionality was probably the easiest part of the assignment as it was mostly normal C coding which i have a lot of experience with.

4 Assignment 3 - Interrupts

4.1 Interrupt Descriptor Table

4.1.1 Theory

The interrupt descriptor table (IDT) is a data structure used by the x86 architecture to handle interrupts and exceptions [12]. It is similarly structured to the global descriptor table and also contains descriptors. Each descriptor contains information about an interrupt or exception handler routine. Because there are a lot of interrupts an operating system can encounter the IDT contains more descriptors than the GDT with a total of 256 descriptors. When an interrupt or exception occurs the processor used the IDT to find the appropriate handler routine to handle the interrupt [12].

4.1.2 Interrupts & exceptions

Interrupts and exceptions are events that disrupt the normal flow of execution within the processor [15]. These events are unexpected and need to be handled before the processor can go back to its normal flow of execution. Interrupts are usually generated by hardware devices such as the mouse or keyboard and are used to signal the processor that an event requires immediate attention. Exceptions on the other hand are triggered by the processor itself in response to an abnormal condition or error that occurs during the normal flow of execution. Both interrupts and exceptions temporarily take control of the processor and make it execute a predefined routine called an interrupt handler or an exception handler. After this routine has been completed the program starts executing from where it left off. Interrupts and exceptions ensure that hardware events are handled promptly, and errors are caught and dealt with [15].

4.1.3 Descriptors in the Interrupt Descriptor Table

Unlike the global descriptor table, the entries in the interrupt descriptor table are not called descriptors [16]. Instead, they are called gates and can either be interrupt gates, task gates, or trap gates. Interrupt gates are used to specify an interrupt service routine

(ISR), or an interrupt request (IRQ), task gates are used for hardware task switching, and trap gates are used to handle exceptions. There are 256 possible interrupt numbers and therefore 256 possible entries within the IDT. Like the GDT, every entry in the IDT contains the same attributes [16]. The attributes for a gate in the IDT are:

- **Base Address:** The base address of the interrupt handler which is the memory address where the interrupt handler's code starts executing. This value is usually split into two parts base low, and base high to fit into the IDT [16].
- **Segment selector:** This is a value that references which segment in the GDT the ISR code resides. In most cases, this segment is the code segment [16].
- **Always0:** A field that is reserved and always set to zero. This is a placeholder to ensure that the IDT has the correct size and works on all processors [16].
- **Flags:** This is an 8-bit value that contains various attributes describing the gate. Examples of these attributes are the type of gate (interrupt, trap, or task), the privilege level of the gate, and whether or not the gate is present in the processor's memory [16].

Figure 9-3. 80386 IDT Gate Descriptors

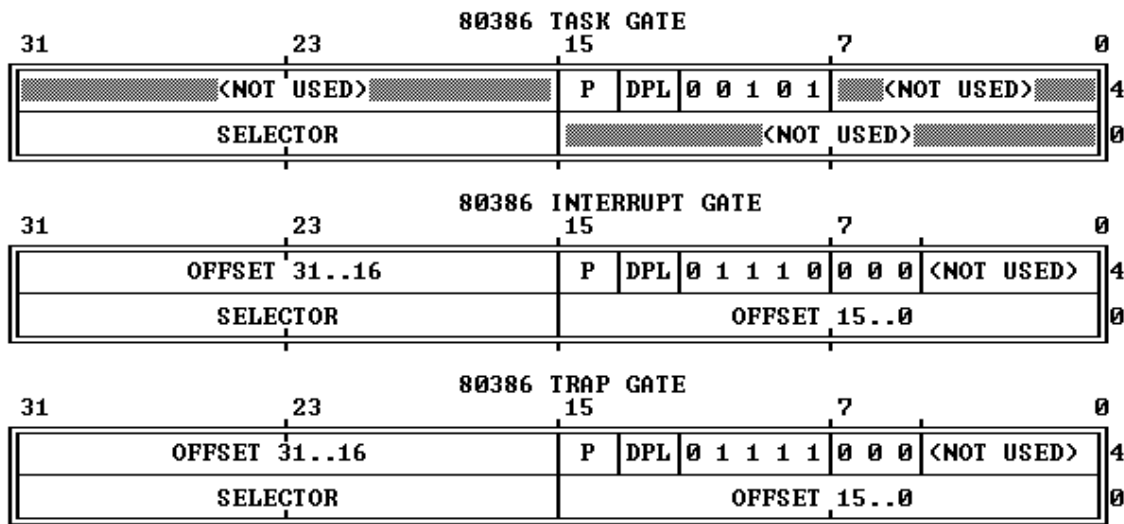


Figure 8: Examples of entries in the Interrupt Descriptor Table[17]

4.1.4 idt.h

My implementation of the IDT is based on the operating system guide by James Molloy [12].

```
1  #ifndef IDT_H
2  #define IDT_H
3
4  #include <stdint.h>
5
6  #define IDT_ENTRIES 256
7
8  struct idt_entry_t
9  {
10     uint16_t base_low;
11     uint16_t sel;
12     uint8_t  always0;
13     uint8_t  flags;
14     uint16_t base_high;
15 } __attribute__((packed));
16
17 struct idt_ptr_t
18 {
19     uint16_t limit;
20     uint32_t base;
21 } __attribute__((packed));
22
23 void idt_set_gate(uint8_t num, uint32_t base, uint16_t sel, uint8_t flags);
24 void init_idt();
25
26 #endif
```

My implementation of the IDT is almost identical to my implementation of the GDT. The `idt.h` header file starts by defining the number of entries in the IDT as 256. There are 256 possible interrupt numbers, so we have to define all 256 entries even if we are not planning on using them [12]. Then a struct called `idt_entry_t` is defined. This struct represents one entry (also known as a gate descriptor) in the IDT, and it contains the fields corresponding to the different components in an IDT entry. The fields in the struct are:

- **base_low:** A 16-bit value that corresponds to the lower 16 bits (bits 0-15) of the 32-bit base address of the interrupt or exception handler routine.

- **sel:** A 16-bit value that represents the segment selector. This value is used to load the appropriate segment into the code segment register whenever an interrupt or exception is triggered.
- **always0:** An 8-bit value that is as the name implies is always set to 0. This value exists to maintain compatibility with processors that expect gate descriptors to be 64 bits long.
- **flags:** An 8-bit value that contains bit flags that describes various attributes of the interrupt gate.
- **base_high:** A 16-bit value that corresponds to the upper 16 bits(bit 16-31) of the 32-bit base address of the interrupt or exception handler routine.

Then another struct called `idt_ptr_t` is defined. Similarly to the GDT, this is a struct that informs the processor where to find my IDT. This struct contains the size and base address of the entire IDT. The fields in the struct are:

- **Limit:** A 16-bit value that represents the size of the IDT in bytes.
- **Base:** A 32-bit value corresponding to the starting address of the IDT in memory.

Lastly, the functions `idt_set_gate` and `init_idt` are declared.

4.1.5 idt.cpp

```

1  #include "idt.h"
2  #include "ports.h"
3
4  extern "C"
5  {
6      #include "memory.h"
7      extern void idt_flush(uint32_t idt_ptr);
8
9      extern void isr0();
10     extern void isr1();
11     extern void isr2();
12     extern void isr3();
13     extern void isr4();
14     extern void isr5();

```



```

15     extern void isr6();
16     extern void isr7();
17     extern void isr8();
18     extern void isr9();
19     extern void isr10();
20     extern void isr11();
21     extern void isr12();
22     extern void isr13();
23     extern void isr14();
24     extern void isr15();
25     extern void isr16();
26     extern void isr17();
27     extern void isr18();
28     extern void isr19();
29     extern void isr20();
30     extern void isr21();
31     extern void isr22();
32     extern void isr23();
33     extern void isr24();
34     extern void isr25();
35     extern void isr26();
36     extern void isr27();
37     extern void isr28();
38     extern void isr29();
39     extern void isr30();
40     extern void isr31();
41     extern void irq0();
42     extern void irq1();
43     extern void irq2();
44     extern void irq3();
45     extern void irq4();
46     extern void irq5();
47     extern void irq6();
48     extern void irq7();
49     extern void irq8();
50     extern void irq9();
51     extern void irq10();
52     extern void irq11();
53     extern void irq12();
54     extern void irq13();
55     extern void irq14();
56     extern void irq15();
57 }
58
59 struct idt_entry_t idt_entries[IDT_ENTRIES];
60 struct idt_ptr_t    idt_ptr;
61
62 void idt_set_gate(uint8_t num, uint32_t base, uint16_t sel, uint8_t flags)

```

```

63 {
64     idt_entries[num].base_low = base & 0xFFFF;
65     idt_entries[num].base_high = (base >> 16) & 0xFFFF;
66
67     idt_entries[num].sel = sel;
68     idt_entries[num].always0 = 0;
69
70     idt_entries[num].flags = flags /* | 0x60*/;
71 }
72
73 void init_idt()
74 {
75     idt_ptr.limit = sizeof(idt_entry_t) * IDT_ENTRIES - 1;
76     idt_ptr.base = (uint32_t)&idt_entries;
77
78     my_memset(&idt_entries, 0, sizeof(idt_entry_t) * IDT_ENTRIES);
79
80     idt_set_gate(0, (uint32_t)isr0, 0x08, 0x8E);
81     idt_set_gate(1, (uint32_t)isr1, 0x08, 0x8E);
82     idt_set_gate(2, (uint32_t)isr2, 0x08, 0x8E);
83     idt_set_gate(3, (uint32_t)isr3, 0x08, 0x8E);
84     idt_set_gate(4, (uint32_t)isr4, 0x08, 0x8E);
85     idt_set_gate(5, (uint32_t)isr5, 0x08, 0x8E);
86     idt_set_gate(6, (uint32_t)isr6, 0x08, 0x8E);
87     idt_set_gate(7, (uint32_t)isr7, 0x08, 0x8E);
88     idt_set_gate(8, (uint32_t)isr8, 0x08, 0x8E);
89     idt_set_gate(9, (uint32_t)isr9, 0x08, 0x8E);
90     idt_set_gate(10, (uint32_t)isr10, 0x08, 0x8E);
91     idt_set_gate(11, (uint32_t)isr11, 0x08, 0x8E);
92     idt_set_gate(12, (uint32_t)isr12, 0x08, 0x8E);
93     idt_set_gate(13, (uint32_t)isr13, 0x08, 0x8E);
94     idt_set_gate(14, (uint32_t)isr14, 0x08, 0x8E);
95     idt_set_gate(15, (uint32_t)isr15, 0x08, 0x8E);
96     idt_set_gate(16, (uint32_t)isr16, 0x08, 0x8E);
97     idt_set_gate(17, (uint32_t)isr17, 0x08, 0x8E);
98     idt_set_gate(18, (uint32_t)isr18, 0x08, 0x8E);
99     idt_set_gate(19, (uint32_t)isr19, 0x08, 0x8E);
100    idt_set_gate(20, (uint32_t)isr20, 0x08, 0x8E);
101    idt_set_gate(21, (uint32_t)isr21, 0x08, 0x8E);
102    idt_set_gate(22, (uint32_t)isr22, 0x08, 0x8E);
103    idt_set_gate(23, (uint32_t)isr23, 0x08, 0x8E);
104    idt_set_gate(24, (uint32_t)isr24, 0x08, 0x8E);
105    idt_set_gate(25, (uint32_t)isr25, 0x08, 0x8E);
106    idt_set_gate(26, (uint32_t)isr26, 0x08, 0x8E);
107    idt_set_gate(27, (uint32_t)isr27, 0x08, 0x8E);
108    idt_set_gate(28, (uint32_t)isr28, 0x08, 0x8E);
109    idt_set_gate(29, (uint32_t)isr29, 0x08, 0x8E);
110    idt_set_gate(30, (uint32_t)isr30, 0x08, 0x8E);

```

```

111     idt_set_gate(31, (uint32_t)isr31, 0x08, 0x8E);
112
113     outb(0x20, 0x11);
114     outb(0xA0, 0x11);
115     outb(0x21, 0x20);
116     outb(0xA1, 0x28);
117     outb(0x21, 0x04);
118     outb(0xA1, 0x02);
119     outb(0x21, 0x01);
120     outb(0xA1, 0x01);
121     outb(0x21, 0x0);
122     outb(0xA1, 0x0);
123
124     idt_set_gate(32, (uint32_t)irq0, 0x08, 0x8E);
125     idt_set_gate(33, (uint32_t)irq1, 0x08, 0x8E);
126     idt_set_gate(34, (uint32_t)irq2, 0x08, 0x8E);
127     idt_set_gate(35, (uint32_t)irq3, 0x08, 0x8E);
128     idt_set_gate(36, (uint32_t)irq4, 0x08, 0x8E);
129     idt_set_gate(37, (uint32_t)irq5, 0x08, 0x8E);
130     idt_set_gate(38, (uint32_t)irq6, 0x08, 0x8E);
131     idt_set_gate(39, (uint32_t)irq7, 0x08, 0x8E);
132     idt_set_gate(40, (uint32_t)irq8, 0x08, 0x8E);
133     idt_set_gate(41, (uint32_t)irq9, 0x08, 0x8E);
134     idt_set_gate(42, (uint32_t)irq10, 0x08, 0x8E);
135     idt_set_gate(43, (uint32_t)irq11, 0x08, 0x8E);
136     idt_set_gate(44, (uint32_t)irq12, 0x08, 0x8E);
137     idt_set_gate(45, (uint32_t)irq13, 0x08, 0x8E);
138     idt_set_gate(46, (uint32_t)irq14, 0x08, 0x8E);
139     idt_set_gate(47, (uint32_t)irq15, 0x08, 0x8E);
140
141     idt_flush((uint32_t)&idt_ptr);
142 }

```

The `idt.cpp` file starts by getting the external function `idt_flush` from the `idt.asm` file. Then it declares the external function names implemented in `interrupts.asm` for the interrupt service routines (ISR) and interrupts request handlers (IRQ), more on these later. These extern directives let me access the addresses of my ASM ISR handlers [12]. I declare 32 ISR directives and 16 IRQ directives. Then I initialize an array of 256 instances of the struct `idt_entry_t` and one instance of the struct `idt_ptr_t`.

idt_set_gate:

I then set up the function `idt_set_gate`. This function is used for setting the values of an entry in the IDT. This is done by populating the fields in the `idt_entries` array. The

function parameters are:

- **num:** An 8-bit value that represents the index of the IDT entry that will be modified.
- **base:** A 32-bit value that represents the base address of the ISR or IRQ for this entry.
- **sel:** A 16-bit value that represents the segment selector which points to the segment that contains the appropriate interrupt handler.
- **flags:** An 8-bit value that represents the access flags that defines the attributes of the interrupt gate.

The attributes are set as followed:

- **base_low:** is set to the lower 16 bits (0-15) of the base address of the interrupt handler.
- **base_high:** is set to higher 16 bits (16-31) of the base address of the interrupt handler.
- **sel:** Is set to the value of the segment selector stored in the sel parameter.
- **always0:** Is set to 0 as it is reserved, and must be set to zero for the IDT entry to work.
- **flags:** Is set to the access flags stored in the flags parameter.

init_idt:

Lastly, I set up the function `init_idt` which is the function responsible for initializing the IDT. The function starts by setting the limit and base values of the `idt_ptr`. The limit is set to the size of one entry times the number of entries - 1 for zero indexing, while the base address is set to the address of the start of the `idt_entries` array which is the start of the entire IDT. Then the function `my_memset` is called which ensures that all the entries are initially empty. I then set the attributes for the first 32 interrupt service routine interrupt handlers using the `idt_set_gate` function. To avoid conflicts between the ISRs and IRQs the programmable interrupt controller(PIC) has to be remapped. This is done by the function `outb`:

```

1 void outb(uint16_t port, uint8_t value)
2 {
3     asm volatile ("outb %1, %0" : : "dN" (port), "a" (value));
4 }

```

This function writes a byte of data to a specified I/O port, which is how the PIC is remapped. Then the function sets the attributes for the 16 interrupt request handlers. Lastly the external function `idt_flush` is called which loads the interrupt descriptor table into the processor's memory.

4.1.6 idt.asm

```

1 [[GLOBAL idt_flush]
2
3 idt_flush:
4     mov eax, [esp+4]
5     lidt [eax]
6     ret

```

The only purpose of `idt.asm` file is to set up the function `idt_flush` to be used in the `idt.cpp` file. It starts by declaring the function as global so it can be used by other files. Then I declare the actual function `idt_flush`. This function takes the pointer to our IDT from the `idt.cpp` file and moves it into the `eax` register. The value `esp+4` contains the address of the IDT pointer passed as an argument. The function `lidt [eax]` loads the IDT pointer into the IDT register. This then loads my self-written IDT into the processor's Interrupt Descriptor Table Register (IDTR) which activates it.

4.1.7 Functionality

I can test that my IDT works by calling a random interrupt in code. Note that this requires the code necessary to handle interrupts which I will get to in a later section. For now this is just a proof that my IDT has been setup correctly. I will use the following code to trigger interrupt number 5:

```
1 __asm__("int $0x5");
```

Which gives me this output:



Figure 9: Proof of a working IDT

4.2 Interrupt Service Routines & Interrupt Requests

4.2.1 Interrupt Service Routines

Interrupt service routines (ISR) are functions that respond to a specific interrupt or exception [18]. For every specific interrupt, there is a specific ISR that will be executed in response. This ISR function is atomic meaning that when it executes it cannot be blocked. Whenever an interrupt occurs the CPU saves the program's current state and executes the corresponding ISR. This ISR then temporarily suspends the normal flow of execution of the program so it can freely handle the interrupt. After the ISR has performed the necessary tasks to handle the interrupt the control returns to the interrupted program that can now continue its normal flow of execution [18].

4.2.2 Interrupt Requests

An interrupt request (IRQ) is a signal sent to the computer's processor by a hardware device or software component to signal that an event requires immediate attention [19]. Once this signal is sent the processor executes an interrupt service routine. The processor then temporarily halts its current execution, allowing the device to run its operation. Once this operation completes the processor starts executing from where it left off. Examples of IRQs are whenever a key on the keyboard is pressed, or the mouse is moved. There are a total of 16 IRQs numbered from 0 to 15 which corresponds to different parts of the computer hardware. This is also the privilege level of each IRQ with IRQ0 having the highest priority and IRQ15 having the lowest. For example, IRQ0 is the system timer which has the highest priority while IRQ15 is the secondary IDE controller which has the lowest priority. The priority level is important for when multiple IRQs are sent at once and the processor has to choose which IRQ to handle first [19].

4.2.3 isr.h

My implementation of ISR's and IRQ's are based on the operating system guide by James Molloy [10]-

```
1  #ifndef ISR_H
2  #define ISR_H
3
4  #include <stdint>
5
6  #define ISR0 0
7  #define ISR1 1
8  #define ISR2 2
9  #define ISR3 3
10 #define ISR4 4
11 #define ISR5 5
12 #define ISR6 6
13 #define ISR7 7
14 #define ISR8 8
15 #define ISR9 9
16 #define ISR10 10
17 #define ISR11 11
18 #define ISR12 12
19 #define ISR13 13
20 #define ISR14 14
```

```

21  #define ISR15 15
22  #define ISR16 16
23  #define ISR17 17
24  #define ISR18 18
25  #define ISR19 19
26  #define ISR20 20
27  #define ISR21 21
28  #define ISR22 22
29  #define ISR23 23
30  #define ISR24 24
31  #define ISR25 25
32  #define ISR26 26
33  #define ISR27 27
34  #define ISR28 28
35  #define ISR29 29
36  #define ISR30 30
37  #define ISR31 31
38  #define IRQ0 32
39  #define IRQ1 33
40  #define IRQ2 34
41  #define IRQ3 35
42  #define IRQ4 36
43  #define IRQ5 37
44  #define IRQ6 38
45  #define IRQ7 39
46  #define IRQ8 40
47  #define IRQ9 41
48  #define IRQ10 42
49  #define IRQ11 43
50  #define IRQ12 44
51  #define IRQ13 45
52  #define IRQ14 46
53  #define IRQ15 47
54
55  typedef struct registers
56  {
57      uint32_t ds;
58      uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
59      uint32_t int_no, err_code;
60      uint32_t eip, cs, eflags, useresp, ss;
61  } registers_t;
62
63  typedef void (*isr_t)(registers_t);
64  void register_interrupt_handler(uint8_t n, isr_t handler);
65
66  #endif

```


This code defines a header file for managing interrupt service routines and interrupt requests. It starts by defining constants for 32 ISRs (ISR0-ISR31) and 16 IRQs (IRQ0-IRQ15) using macro definitions. These are defined from interrupt number 0 to interrupt number 47. Then a struct called `register` is declared which stores the processor's state every time an interrupt is triggered. This struct holds the value of the data segment selector, the various general-purpose registers values, the interrupt number and error code, and other critical register information. Then a new datatype `isr_t` is defined as a function pointer. This data type represents a function that will handle an ISR or IRQ. This function takes a single instance of the `register` struct as a parameter. Lastly, the header file declares the function `register_interrupt_handler` which is a function that allows the program to specify custom behavior to handle each interrupt.

4.2.4 isr.cpp

```
1  #include "isr.h"
2  #include "ports.h"
3  #include "../drivers/screen.h"
4
5  extern "C"
6  {
7      void isr_handler(registers_t regs);
8      void irq_handler(registers_t regs);
9  }
10
11  isr_t interrupt_handlers[256];
12
13  char *exception_messages[] = {
14      "Division by zero exception",
15      "Debug exception",
16      "Non maskable interrupt",
17      "Breakpoint exception",
18      "Into detected overflow",
19      "Out of bounds exception",
20      "Invalid opcode exception",
21      "No coprocessor exception",
22      "Double fault",
23      "Coprocessor segment overrun",
24      "Bad TSS",
25      "Segment not present",
26      "Stack fault",
27      "General protection fault",
```

```

28     "Page fault",
29     "Unknown interrupt exception",
30     "Coprocesor fault",
31     "Alignment check exception",
32     "Machine check exception",
33     "Reserved",
34     "Reserved",
35     "Reserved",
36     "Reserved",
37     "Reserved",
38     "Reserved",
39     "Reserved",
40     "Reserved",
41     "Reserved",
42     "Reserved",
43     "Reserved",
44     "Reserved",
45     "Reserved"
46 };
47
48 void register_interrupt_handler(uint8_t n, isr_t handler)
49 {
50     interrupt_handlers[n] = handler;
51 }
52
53 void isr_handler(registers_t regs)
54 {
55     printf("Recieved interrupt %d: %s\n", regs.int_no,
56     ↪ exception_messages[regs.int_no]);
57 }
58
59 void irq_handler(registers_t regs)
60 {
61     if (regs.int_no >= 40)
62     {
63         outb(0xA0, 0x20);
64     }
65     outb(0x20, 0x20);
66
67     if (interrupt_handlers[regs.int_no] != 0)
68     {
69         isr_t handler = interrupt_handlers[regs.int_no];
70         handler(regs);
71     }
72 }

```

The `isr.cpp` file starts by declaring the `isr_handler`, and `irq_handler` as extern functions so they can later be used in the `interrupt.asm` file to handle interrupts. Next, a function pointer array called `interrupt_handler` is declared. This is an array of 256 ISR or IRQ handlers, one for each possible interrupt. Then another array called `exception_messages` is declared. This array holds the descriptions of the most commonly occurring exceptions in order of their interrupt number so their description can be printed whenever they occur.

To be able to handle interrupts in my program I need some functions.

register_interrupt_handler:

`Register_interrupt_handler` is a function to register a handler function to a specific interrupt number. It takes the number of the interrupt and the function that will handle said interrupt as its parameters. Then it sets the value at the interrupts position in the `interrupt_handler` array to that function essentially storing the interrupt number and its handler as a pair.

isr_handler:

The `isr_handler` function is the handler for ISR that occurs whenever an ISR interrupt happens. It takes an instance of the `register` struct as its parameter which contains information about the interrupt. The function simply prints the interrupt number alongside its corresponding message.

irq_handler:

The last function in the `isr.cpp` file is the `irq_handler` function which handles IRQs. Similarly to the `isr_handler` function this function also takes an instance of the `register` struct as its parameter. The function first checks if the interrupt number is higher or equal to 40. This is because the programmable interrupt controller (PIC) has two chips master PIC and slave PIC. The master PIC handles interrupts 32-39 while the slave PIC handles interrupts 40-47. If the interrupt belongs to the slave PIC a reset signal is sent to the slave PICs' command port to let it acknowledge the interrupt and inform it that it can continue processing other interrupts. If the interrupt belongs to the master PIC this signal is sent to the master PICs' command port instead. Lastly, the function checks if the IRQ that is being handled has a corresponding IRQ handler function. If it does this function is executed.

4.2.5 interrupt.asm

```
1  %macro ISR_NOERRCODE 1
2  global isr%1
3  isr%1:
4      ;cli
5      push byte 0
6      push %1
7      jmp isr_common_stub
8  %endmacro
9
10 %macro ISR_ERRCODE 1
11 global isr%1
12 isr%1:
13     ;cli
14     push %1
15     jmp isr_common_stub
16 %endmacro
17
18 ISR_NOERRCODE 0
19 ISR_NOERRCODE 1
20 ISR_NOERRCODE 2
21 ISR_NOERRCODE 3
22 ISR_NOERRCODE 4
23 ISR_NOERRCODE 5
24 ISR_NOERRCODE 6
25 ISR_NOERRCODE 7
26 ISR_ERRCODE 8
27 ISR_NOERRCODE 9
28 ISR_ERRCODE 10
29 ISR_ERRCODE 11
30 ISR_ERRCODE 12
31 ISR_ERRCODE 13
32 ISR_ERRCODE 14
33 ISR_NOERRCODE 15
34 ISR_NOERRCODE 16
35 ISR_ERRCODE 17
36 ISR_NOERRCODE 18
37 ISR_NOERRCODE 19
38 ISR_NOERRCODE 20
39 ISR_NOERRCODE 21
40 ISR_NOERRCODE 22
41 ISR_NOERRCODE 23
42 ISR_NOERRCODE 24
43 ISR_NOERRCODE 25
44 ISR_NOERRCODE 26
45 ISR_NOERRCODE 27
```

```

46 ISR_NOERRCODE 28
47 ISR_NOERRCODE 29
48 ISR_ERRCODE 30
49 ISR_NOERRCODE 31
50
51 extern isr_handler
52
53 isr_common_stub:
54     pusha
55
56     mov ax, ds
57     push eax
58
59     mov ax, 0x10
60     mov ds, ax
61     mov es, ax
62     mov fs, ax
63     mov gs, ax
64
65     call isr_handler
66
67     pop eax
68     mov ds, ax
69     mov es, ax
70     mov fs, ax
71     mov gs, ax
72
73     popa
74     add esp, 8
75     sti
76     iret
77
78 %macro IRQ 2
79 global irq%1
80 irq%1:
81     ;cli
82     push byte 0
83     push byte %2
84     jmp irq_common_stub
85 %endmacro
86
87 IRQ 0, 32
88 IRQ 1, 33
89 IRQ 2, 34
90 IRQ 3, 35
91 IRQ 4, 36
92 IRQ 5, 37
93 IRQ 6, 38

```

```

94  IRQ    7,    39
95  IRQ    8,    40
96  IRQ    9,    41
97  IRQ   10,    42
98  IRQ   11,    43
99  IRQ   12,    44
100  IRQ   13,    45
101  IRQ   14,    46
102  IRQ   15,    47
103
104  extern irq_handler
105
106  irq_common_stub:
107      pusha
108
109      mov ax, ds
110      push eax
111
112      mov ax, 0x10
113      mov ds, ax
114      mov es, ax
115      mov fs, ax
116      mov gs, ax
117
118      call irq_handler
119
120      pop ebx
121      mov ds, bx
122      mov es, bx
123      mov fs, bx
124      mov gs, bx
125
126      popa
127      add esp, 8
128      sti
129      iret

```

ISR:

The interrupt.asm file starts by declaring two macros `ISR_NOERRCODE` and `ISR_ERRCODE`. These macros define ISRs with and without error codes, as some ISRs have an error message while some do not. These macros accept a single parameter which is the ISR interrupt number. Each ISR has a global label which is `ISR` followed by their corresponding number. The macro then pushes an error code if applicable, and the ISR number onto the stack. Lastly, the ISR macro jumps to the `isr_common_stub` part of the file.

Next in the code, we define if the first 32 interrupts have error codes or not to know which macro to use. Then we get to the `isr_common_stub` function. This is a function that is called by all ISRs whenever they get called by the processor. The function saves the processor's state/registers, loads the kernel data segment descriptors, and calls the `isr_handler` function with the current values in the registers as its parameter. Lastly, the function restores the processor state before returning from the interrupt.

IRQ:

We also have to create a macro for defining IRQs. IRQs don't differentiate in having error codes so we only have to make a single macro for defining them. Each IRQ has a global label that can be found with IRQ followed by their number. The macro takes two parameters the IRQ number and the remapped ISR number. Then similarly to the ISR macro, it pushes the error code as well as the ISR number onto the stack before jumping to the `irq_common_stub` function. Next in the code we map the IRQ numbers (0-15) to their respective interrupt/ISR numbers (32-47). Then we get to the `irq_common_stub` function. This is the common IRQ stub called by all IRQs whenever they get executed. It follows the exact same process as the `isr_common_stub` function but calls the `irq_handler` function instead of the `isr_handler` function.

4.2.6 Functionality

I can check that interrupts work in my program by trying to divide by 0 in main. This will keep calling the `isr_handler` function with exception number 0:

```

56 //Handler for ISR's that occurs when an interrupt happens.
57 void isr_handler(registers_t regs)
58 {
59     //Prints the interrupt to the terminal.
60     printf("Recieved interrupt %d: %s\n", regs.int_no, exception_messages[regs.int_no]);
61 }
62
63 //Handler for x86-64 interrupts
64
65
66 Multiboot check went OK. eax=732803074, magic=732803074
67 Recieved interrupt 0: Division by zero exception
68 Recieved interrupt 0: Division by zero exception
69 Recieved interrupt 0: Division by zero exception
70 Recieved interrupt 0: Division by zero exception
71 Recieved interrupt 0: Division by zero exception
72
73

```

Exception has occurred. X
Hit breakpoint 1 at 0x100a37.

QEMU [Paused]

Machine View

Figure 10: Example of exceptions/interrupts working in my program

4.3 Keyboard Logger

4.3.1 keyboard.h

```

1  #ifndef KEYBOARD_H
2  #define KEYBOARD_H
3
4  #define KEYBOARD_DATA_PORT 0x60
5  #define KEYBOARD_STATUS_PORT 0x64
6  #define KEYBOARD_COMMAND_PORT 0x64
7
8  #define SCAN_CODE_KEY_ESC 0x01
9  #define SCAN_CODE_KEY_1 0x02
10 #define SCAN_CODE_KEY_2 0x03
11 #define SCAN_CODE_KEY_3 0x04
12 #define SCAN_CODE_KEY_4 0x05
13 #define SCAN_CODE_KEY_5 0x06
14 #define SCAN_CODE_KEY_6 0x07
15 #define SCAN_CODE_KEY_7 0x08
16 #define SCAN_CODE_KEY_8 0x09
17 #define SCAN_CODE_KEY_9 0x0A
18 #define SCAN_CODE_KEY_0 0x0B
19 #define SCAN_CODE_KEY_MINUS 0x0C
20 #define SCAN_CODE_KEY_EQUAL 0x0D
21 #define SCAN_CODE_KEY_BACKSPACE 0x0E
22 #define SCAN_CODE_KEY_TAB 0x0F

```



```

23 #define SCAN_CODE_KEY_Q 0x10
24 #define SCAN_CODE_KEY_W 0x11
25 #define SCAN_CODE_KEY_E 0x12
26 #define SCAN_CODE_KEY_R 0x13
27 #define SCAN_CODE_KEY_T 0x14
28 #define SCAN_CODE_KEY_Y 0x15
29 #define SCAN_CODE_KEY_U 0x16
30 #define SCAN_CODE_KEY_I 0x17
31 #define SCAN_CODE_KEY_O 0x18
32 #define SCAN_CODE_KEY_P 0x19
33 #define SCAN_CODE_KEY_SQUARE_OPEN_BRACKET 0x1A
34 #define SCAN_CODE_KEY_SQUARE_CLOSE_BRACKET 0x1B
35 #define SCAN_CODE_KEY_ENTER 0x1C
36 #define SCAN_CODE_KEY_LEFT_CTRL 0x1D
37 #define SCAN_CODE_KEY_A 0x1E
38 #define SCAN_CODE_KEY_S 0x1F
39 #define SCAN_CODE_KEY_D 0x20
40 #define SCAN_CODE_KEY_F 0x21
41 #define SCAN_CODE_KEY_G 0x22
42 #define SCAN_CODE_KEY_H 0x23
43 #define SCAN_CODE_KEY_J 0x24
44 #define SCAN_CODE_KEY_K 0x25
45 #define SCAN_CODE_KEY_L 0x26
46 #define SCAN_CODE_KEY_SEMICOLON 0x27
47 #define SCAN_CODE_KEY_SINGLE_QUOTE 0x28
48 #define SCAN_CODE_KEY_ACUTE 0x29
49 #define SCAN_CODE_KEY_LEFT_SHIFT 0x2A
50 #define SCAN_CODE_KEY_BACKSLASH 0x2B
51 #define SCAN_CODE_KEY_Z 0x2C
52 #define SCAN_CODE_KEY_X 0x2D
53 #define SCAN_CODE_KEY_C 0x2E
54 #define SCAN_CODE_KEY_V 0x2F
55 #define SCAN_CODE_KEY_B 0x30
56 #define SCAN_CODE_KEY_N 0x31
57 #define SCAN_CODE_KEY_M 0x32
58 #define SCAN_CODE_KEY_COMMA 0x33
59 #define SCAN_CODE_KEY_DOT 0x34
60 #define SCAN_CODE_KEY_FORESLHASH 0x35
61 #define SCAN_CODE_KEY_RIGHT_SHIFT 0x36
62 #define SCAN_CODE_KEY_ASTERISK 0x37
63 #define SCAN_CODE_KEY_ALT 0x38
64 #define SCAN_CODE_KEY_SPACE 0x39
65 #define SCAN_CODE_KEY_CAPS_LOCK 0x3A
66 #define SCAN_CODE_KEY_F1 0x3B
67 #define SCAN_CODE_KEY_F2 0x3C
68 #define SCAN_CODE_KEY_F3 0x3D
69 #define SCAN_CODE_KEY_F4 0x3E
70 #define SCAN_CODE_KEY_F5 0x3F

```

```

71 #define SCAN_CODE_KEY_F6 0x40
72 #define SCAN_CODE_KEY_F7 0x41
73 #define SCAN_CODE_KEY_F8 0x42
74 #define SCAN_CODE_KEY_F9 0x43
75 #define SCAN_CODE_KEY_F10 0x44
76 #define SCAN_CODE_KEY_NUM_LOCK 0x45
77 #define SCAN_CODE_KEY_SCROLL_LOCK 0x46
78 #define SCAN_CODE_KEY_HOME 0x47
79 #define SCAN_CODE_KEY_UP 0x48
80 #define SCAN_CODE_KEY_PAGE_UP 0x49
81 #define SCAN_CODE_KEY_KEYPAD_MINUS 0x4A
82 #define SCAN_CODE_KEY_LEFT 0x4B
83 #define SCAN_CODE_KEY_KEYPAD_5 0x4C
84 #define SCAN_CODE_KEY_RIGHT 0x4D
85 #define SCAN_CODE_KEY_KEYPAD_PLUS 0x4E
86 #define SCAN_CODE_KEY_END 0x4F
87 #define SCAN_CODE_KEY_DOWN 0x50
88 #define SCAN_CODE_KEY_PAGE_DOWN 0x51
89 #define SCAN_CODE_KEY_INSERT 0x52
90 #define SCAN_CODE_KEY_DELETE 0x53
91 #define SCAN_CODE_KEY_F11 0x57
92 #define SCAN_CODE_KEY_F12 0x58
93
94 void init_keyboard();
95
96 #endif

```

The keyboard.h file is mainly for defining keyboard-related constants as well as scancodes for the different keys in a QWERTY keyboard layout. The file starts by defining three keyboard port addresses. These are the keyboard data port, keyboard status port, and keyboard command port. These ports are used to communicate with the keyboard controller. The rest of the definitions are for defining all the scancode constants needed for a QWERTY keyboard layout. We define the port for every key on the keyboard so we can use them later by their name instead of their hexadecimal value. All the keyboard scancodes can be found here: [scancodes](#). Lastly, the header file declares the `init_keyboard` function.

4.3.2 keyboard.cpp

```

1  #include "keyboard.h"
2  #include "isr.h"
3  #include "ports.h"
4  #include "screen.h"
5
6  extern "C"
7  {
8      #include "strings.h"
9  }
10
11 static int caps_lock = 0;
12 static int shift_pressed = 0;
13 static int altgr_pressed = 0;
14
15 char scan_code_chars[128] = {
16     0, 27, '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', '=', '\b',
17     '\t', 'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', '[', ']', '\n',
18     0, 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', '\'', '`', 0,
19     '\\', 'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.', '/', 0, '*', 0, ' ',
20     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, '-', 0, 0, 0, '+', 0, 0,
21     0, 0, 0, 0, 0, 0, 0, 0, 0
22 };
23
24 static int get_scancode()
25 {
26     int scancode = 0;
27
28     while(1)
29     {
30         if((inb(KEYBOARD_STATUS_PORT) & 1) != 0)
31         {
32             scancode = inb(KEYBOARD_DATA_PORT);
33             break;
34         }
35     }
36     return scancode;
37 }
38
39 char shift_alterate_chars(char ch)
40 {
41     switch(ch) {
42         case '1': return '!';
43         case '2': return '"';
44         case '3': return '#';
45         case '4': return '$';
46         case '5': return '%';
47         case '6': return '&';

```

```

48     case '7': return '/';
49     case '8': return '(';
50     case '9': return ')';
51     case '0': return '=';
52     case '+': return '?';
53     case '\\': return '^';
54     case '\': return '*';
55     case '§': return '½';
56     case 'å': return 'Å';
57     case '¨': return '^';
58     case 'æ': return 'Æ';
59     case 'ø': return 'Ø';
60     case '-': return '_';
61     case ',': return ';';
62     case '.': return ':';
63     case '/': return '|';
64     default: return to_upper_char(ch);
65 }
66 }
67
68 char altgr_alternate_chars(char ch)
69 {
70     switch(ch) {
71         case '2': return '@';
72         case '3': return '#';
73         case '4': return '$';
74         case '5': return '%';
75         case '6': return '{';
76         case '7': return '[';
77         case '8': return ']';
78         case '9': return '}';
79         case '0': return '\\';
80         case '+': return '~';
81         case '<': return '|';
82         case 'm': return 'µ';
83         default: return ch;
84     }
85 }
86
87 static void keyboard_handler(registers_t regs)
88 {
89     int scancode;
90     char ch = 0;
91
92     scancode = get_scancode();
93
94     if(scancode & 0x80)
95     {

```

```

96         scancode &= ~0x80;
97
98         if (scancode == SCAN_CODE_KEY_LEFT_SHIFT || scancode ==
↪ SCAN_CODE_KEY_RIGHT_SHIFT)
99         {
100             shift_pressed = 0;
101         }
102         else if(scancode == SCAN_CODE_KEY_ALT)
103         {
104             altgr_pressed = 0;
105         }
106     }
107     else
108     {
109         switch(scancode)
110         {
111             case SCAN_CODE_KEY_CAPS_LOCK:
112                 caps_lock = !caps_lock;
113                 break;
114
115             case SCAN_CODE_KEY_ENTER:
116                 printf("\n");
117                 break;
118
119             case SCAN_CODE_KEY_TAB:
120                 printf("\t");
121                 break;
122
123             case SCAN_CODE_KEY_LEFT_SHIFT:
124             case SCAN_CODE_KEY_RIGHT_SHIFT:
125                 shift_pressed = 1;
126                 break;
127
128             case SCAN_CODE_KEY_ALT:
129                 altgr_pressed = 1;
130                 break;
131
132             case SCAN_CODE_KEY_BACKSPACE:
133                 backspace();
134                 break;
135
136             case SCAN_CODE_KEY_UP:
137                 scroll_up();
138                 break;
139
140             case SCAN_CODE_KEY_DOWN:
141                 scroll_down();
142                 break;

```

```

143
144     default:
145         ch = scan_code_chars[scancode];
146
147         if(caps_lock)
148         {
149             if(shift_pressed)
150             {
151                 ch = shift_alternate_chars(ch);
152             }
153             else if(altgr_pressed)
154             {
155                 ch = altgr_alternate_chars(ch);
156             }
157             else
158             {
159                 ch = to_upper_char(ch);
160             }
161         }
162         else
163         {
164             if(shift_pressed)
165             {
166                 if(isalpha(ch))
167                 {
168                     ch = to_upper_char(ch);
169                 }
170                 else
171                 {
172                     ch = shift_alternate_chars(ch);
173                 }
174             }
175             else if(altgr_pressed)
176             {
177                 ch = altgr_alternate_chars(ch);
178             }
179         }
180         printf("%c", ch);
181     }
182 }
183
184
185 void init_keyboard()
186 {
187     register_interrupt_handler(IRQ1, keyboard_handler);
188 }

```

The `keyboard.cpp` file starts by declaring three variables `caps_lock`, `shift_pressed`, `altgr_pressed`. These variables keep track of whether the caps lock is on or off, and whether or not the shift and altgr buttons are currently being pressed. I then define a lookup table called `scan_code_chars` where each index corresponds to a scancode defined in `keyboard.h`.

get_scancode:

The first function in the file is the `get_scancode` function. This function gets the scancode of the key that gets pressed on the keyboard. The function enters an infinite loop and keeps checking if the keyboard has sent a scancode by checking the first bit of the keyboard status port. When it finds that a scancode has been sent it returns it.

shift_alternate_chars & altgr_alternate_chars:

The next two functions `shift_alternate_chars` and `altgr_alternate_chars` are used to get the alternate version of certain keys on the keyboard. These functions are practically identical and use a switch statement to return the alternate characters of a keyboard key if the shift or altgr button is currently being pressed.

keyboard_handler:

Now we have come to the actual `keyboard_handler` function. This is a handler function or in other words a function that gets mapped together with an ISR or IRQ. In this case, the `keyboard_handler` function gets mapped to IRQ1 as IRQ1 is assigned to the keyboard (PS/2). The function starts by declaring the variables `scancode` to hold the scancode and `ch` to hold the scancode character. The `scancode` is then set using the `get_scancode` function. The first if statement checks if the 0x80 bit is in the scancode. The 0x80 bit is in the scancode if the key has been released. If the key that has been released (has the 0x80 bit) is the shift key the `shift_pressed` is set to 0 (false) or if it is the altgr key the `altgr_pressed` is set to 0 (false). If the scancode does not have the 0x80 bit that means the key has just been pressed and the scancode goes through a switch statement. This switch statement checks if any special key has been pressed. The caps lock key turns the `caps_lock` variable to true or false, the enter key prints a newline, the tab key prints an indentation, the shift key turns `shift_pressed` to true, the alt key turns the `altgr_pressed` to true, the backspace key erases the newest printed character, and the up and down key scrolls up and down. If the key pressed is not a special key we take the scancode through the lookup table and store the character in the `ch` variable. We then

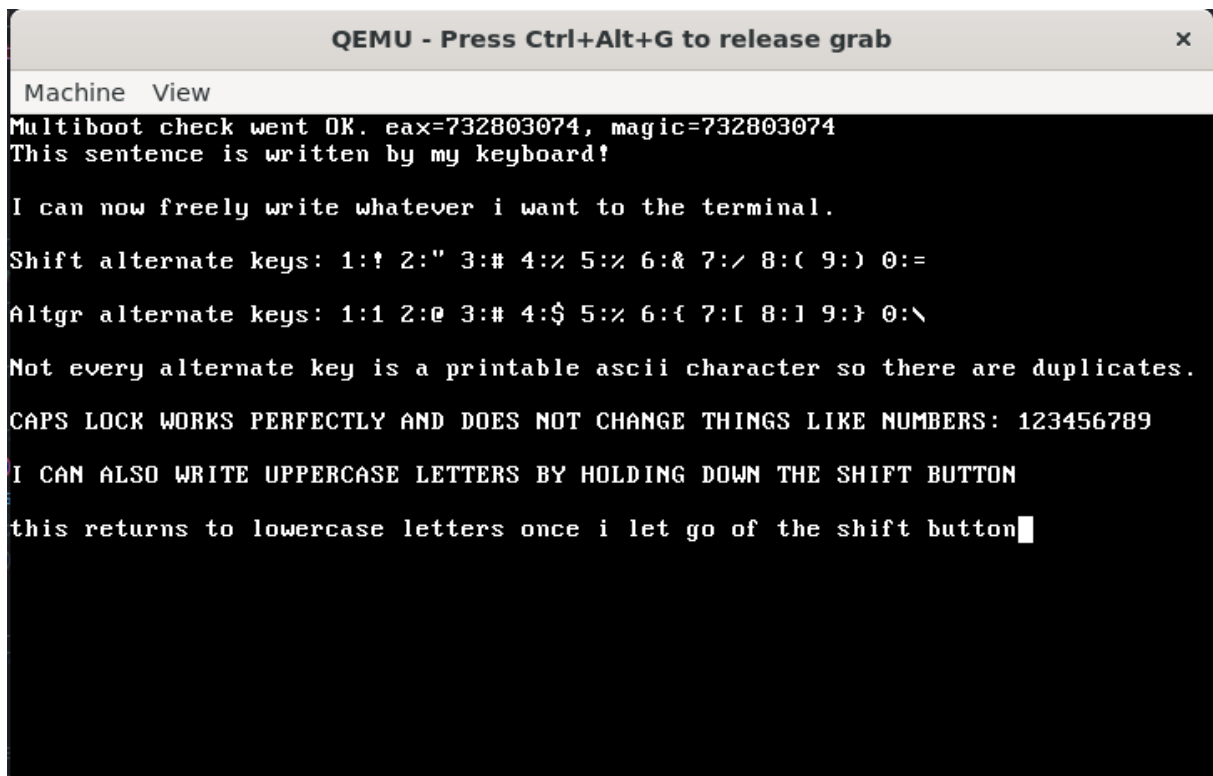
go through some if and else statements that check if the caps lock is on, or if the shift or altgr key is pressed. If this is the case it either changes the character to a capital letter or an alternate character depending on the keys being pressed. Lastly, the function prints the `ch` character to the terminal.

init_keyboard:

The `init_keyboard` function at the bottom of the `keyboard.cpp` file maps the `keyboard_handler` function to IRQ1 using the `register_interrupt_handler` function I created in `isr.cpp`.

4.3.3 Functionality:

After running the `init_keyboard` function in `main` I can now freely type to my terminal using the keyboard:

A screenshot of a QEMU terminal window. The title bar reads "QEMU - Press Ctrl+Alt+G to release grab" with a close button on the right. Below the title bar is a tab labeled "Machine View". The terminal content shows the following text:

```
Multiboot check went OK. eax=732803074, magic=732803074
This sentence is written by my keyboard!

I can now freely write whatever i want to the terminal.
Shift alternate keys: 1:! 2:" 3:# 4:% 5:% 6:& 7:/ 8:( 9:) 0:=
Altgr alternate keys: 1:1 2:@ 3:# 4:$ 5:% 6:{ 7:[ 8:] 9:} 0:\
Not every alternate key is a printable ascii character so there are duplicates.
CAPS LOCK WORKS PERFECTLY AND DOES NOT CHANGE THINGS LIKE NUMBERS: 123456789
I CAN ALSO WRITE UPPERCASE LETTERS BY HOLDING DOWN THE SHIFT BUTTON
this returns to lowercase letters once i let go of the shift button
```

Figure 11: Writing to my terminal using the keyboard

4.4 Approach and challenges

My approach for assignment 3 was to jump straight into the assignment tasks. This is because I had a pretty good understanding of os development at this point. Implementing the IDT was simple as it was almost identical to the implementation of the GDT code wise. Setting up the interrupt handlers were a bit trickier but was also pretty straight forward as James Molloy has a pretty informative guide on the subject [20]. The biggest challenge I had in assignment 3 was the implementation of the keyboard handler as there was not any particularly useful guides on the subject. I had to scour different GitHub repositories to find insperations on how to set up the keyboard logic. Eventually I managed to code the logic needed for my keyboard to work. At this point the next challenge arose. This was being able to get the function to know when I was holding down a button like shift or altgr. I eventually found the 0x80 bit for key release that could keep track of button presses.

5 Assignment 4 - Memory Management, PIT, and Pull Request

5.1 Memory Management

5.1.1 Paging

Paging is a memory management technique used in computers [21]. It is the second memory management technique used in the x86 architecture behind segmentation. The main purpose of paging is to store and retrieve data from secondary storage and use it as main memory. Furthermore, it helps in managing and allocating a computer's physical memory efficiently and effectively. Paging is done by dividing the physical and virtual memory of a computer into fixed-sized chunks which are called pages. Then when a program wants to store or retrieve data it uses the virtual addresses. These addresses consist of the page number and an offset within the page. The operating system then uses a page table to translate this virtual address into the corresponding physical address. The physical address is where the data the program wants to use actually is. The process of translating virtual addresses to physical addresses is called address translation. The process of paging is also called the process of virtual memory. Using this method the computer can be tricked into thinking it has more primary memory than is physically installed on the computer which increases system efficiency and provides memory protection [21].

5.1.2 memory_management.h

My implementation of memory management and paging is heavily based on the files found in `Assignment files` from assignment 4. I struggled with implementing paging on my own and had to resort to using the provided solution.

```
1  #ifndef MEMORY_H
2  #define MEMORY_H
3
4  #include <stdint.h>
5  #include <stddef.h>
6
7  typedef struct {
8      uint8_t status;
```

```

9      uint32_t size;
10 } alloc_t;
11
12 void init_kernel_memory(uint32_t* kernel_end);
13 void print_memory_layout();
14 void free(void *mem);
15 void pfree(void *mem);
16 char* pmalloc(size_t size);
17 void* malloc(size_t size);
18 void *my_memset(void *str, int c, int len);
19
20 void paging_map_virtual_to_phys(uint32_t virt, uint32_t phys);
21 void paging_enable();
22 void init_paging();
23
24 #endif

```

The `memory_management.h` file starts by defining a new datatype called `alloc_t`. This datatype is a struct that represents a memory allocation. It contains two fields: `status` which tells us if the memory block is currently allocated or not, and `size` which tells us how large the memory block is in bytes. This datatype will be used for allocating blocks of memory. Lastly in the header file are the function declarations. I have two `.c` files that corresponds to this header file. These are `memory_management.c` and `paging.c`. The first seven functions are defined in the `memory_management.c` functions and are system wide memory management functions. The three other functions are paging specific and are therefore placed in the `paging.c` file.

5.1.3 `memory_management.c`

```

1  #include "memory_management.h"
2
3  #define MAX_PAGE_ALIGNED_ALLOCS 32
4
5  uint32_t last_alloc = 0;
6  uint32_t heap_begin = 0;
7  uint32_t heap_end = 0;
8  uint32_t pheap_begin = 0;
9  uint32_t pheap_end = 0;
10 uint8_t *pheap_desc = 0;
11 uint32_t memory_used = 0;

```

```

12
13 void init_kernel_memory(uint32_t* kernel_end)
14 {
15     last_alloc = kernel_end + 0x1000;
16     heap_begin = last_alloc;
17     pheap_end = 0x400000;
18     pheap_begin = pheap_end - (MAX_PAGE_ALIGNED_ALLOCS * 4096);
19     heap_end = pheap_begin;
20     my_memset((char *)heap_begin, 0, heap_end - heap_begin);
21     pheap_desc = (uint8_t *)malloc(MAX_PAGE_ALIGNED_ALLOCS);
22     printf("Kernel heap starts at 0x%x\n", last_alloc);
23 }
24
25 void print_memory_layout()
26 {
27     printf("Memory used: %d bytes\n", memory_used);
28     printf("Memory free: %d bytes\n", heap_end - heap_begin - memory_used);
29     printf("Heap size: %d bytes\n", heap_end - heap_begin);
30     printf("Heap start: 0x%x\n", heap_begin);
31     printf("Heap end: 0x%x\n", heap_end);
32     printf("PHeap start: 0x%x\nPHeap end: 0x%x\n", pheap_begin, pheap_end);
33 }
34
35 void free(void *mem)
36 {
37     alloc_t *alloc = (mem - sizeof(alloc_t));
38     memory_used -= alloc->size + sizeof(alloc_t);
39     alloc->status = 0;
40 }
41
42 void pfree(void *mem)
43 {
44     if(mem < pheap_begin || mem > pheap_end) return;
45
46     uint32_t ad = (uint32_t)mem;
47     ad -= pheap_begin;
48     ad /= 4096;
49
50     pheap_desc[ad] = 0;
51 }
52
53 char* pmalloc(size_t size)
54 {
55     for(int i = 0; i < MAX_PAGE_ALIGNED_ALLOCS; i++)
56     {
57         if(pheap_desc[i]) continue;
58         pheap_desc[i] = 1;
59         printf("PAllocated from 0x%x to 0x%x\n", pheap_begin + i*4096, pheap_begin +
↵ (i+1)*4096);

```

```

60     return (char *) (pheap_begin + i*4096);
61 }
62 printf("pmalloc: FATAL: failure!\n");
63 return 0;
64 }
65
66 void* malloc(size_t size)
67 {
68     if(!size) return 0;
69
70     uint8_t *mem = (uint8_t *)heap_begin;
71     while((uint32_t)mem < last_alloc)
72     {
73         alloc_t *a = (alloc_t *)mem;
74         printf("mem=0x%x a={.status=%d, .size=%d}\n", mem, a->status, a->size);
75
76         if(!a->size)
77             goto nalloc;
78         if(a->status) {
79             mem += a->size;
80             mem += sizeof(alloc_t);
81             mem += 4;
82             continue;
83         }
84
85         if(a->size >= size)
86         {
87             a->status = 1;
88             printf("RE:Allocated %d bytes from 0x%x to 0x%x\n", size, mem +
↪ sizeof(alloc_t), mem + sizeof(alloc_t) + size);
89             my_memset(mem + sizeof(alloc_t), 0, size);
90             memory_used += size + sizeof(alloc_t);
91             return (char *) (mem + sizeof(alloc_t));
92         }
93
94         mem += a->size;
95         mem += sizeof(alloc_t);
96         mem += 4;
97     }
98
99     nalloc::;
100     if(last_alloc + size + sizeof(alloc_t) >= heap_end)
101     {
102         printf("Cannot allocate bytes! Out of memory.\n");
103         return;
104     }
105     alloc_t *alloc = (alloc_t *)last_alloc;
106     alloc->status = 1;

```

```

107     alloc->size = size;
108
109     last_alloc += size;
110     last_alloc += sizeof(alloc_t);
111     last_alloc += 4;
112     printf("Allocated %d bytes from 0x%x to 0x%x\n", size, (uint32_t)alloc +
↪     sizeof(alloc_t), last_alloc);
113     memory_used += size + 4 + sizeof(alloc_t);
114     my_memset((char *)((uint32_t)alloc + sizeof(alloc_t)), 0, size);
115     return (char *)((uint32_t)alloc + sizeof(alloc_t));
116 }
117
118 void *my_memset(void *str, int c, int len)
119 {
120     unsigned char* p = str;
121
122     while(len--)
123     {
124         *p++ = (unsigned char)c;
125     }
126     return str;
127 }

```

My `memory_management.c` file starts by defining the max number of page aligned allocations that are allowed as 32. Then seven global variables are declared which are used to manage the kernel's memory. These variables are:

- `last_alloc`: The address of the last memory allocation made in the heap.
- `heap_begin`: The address of the start of the heap.
- `heap_end`: The address of the end of the heap.
- `pheap_begin`: The address of the start of the page-aligned heap.
- `pheap_end`: The address of the end of the page-aligned heap.
- `phead_desc`: An array where each entry represents if a page is occupied or not.
- `memory_used`: A value that keeps track of the total memory that has been allocated.

init_kernel_memory:

The first function in the file is the `init_kernel_memory` which initializes the kernel memory manager. It takes in one parameter `kernel_end` which is the address of the end of the

memory used by the kernel. This function basically initializes the global heap variables I declared above which in turn initializes the kernel memory manager. The variables are set as followed:

- `last_alloc` is set to one page after the end of the kernel.
- `heap_begin` is also set to one page after the end of the kernel which is where the heap begins.
- `pheap_end` is set to the address `0x400000`.
- `pheap_begin` is set to 32 pages before the page-aligned heap ends.
- `heap_end` is set to `pheap_begin` as the heap ends where the page-aligned heap begins. After this the entire heap memory is set to zero using the `my_memset` function.
- `phead_desc` has the memory for the page-aligned heap allocated.

Lastly, the function prints the address where the kernel heap starts.

print_memory_layout:

This function prints out the current state of memory within my operating system. The parts of the memory that is printed is the memory that is in use, the memory that is free, the size of the heap, the start and end of the heap, as well as the start and end of the page-aligned heap. To print these various parts of memory I use the global variables declared and initialized above.

free:

The `free` function is used to free a block of memory, and it takes a void pointer `mem` as its parameter. When you pass a void pointer to this function you are giving it the address of the block of memory that you want to free. By subtracting the size of `alloc_t` from this memory block we get a pointer to a metadata block that contains information about the allocation like its size and status. This memory block is stored as a new `alloc_t` pointer datatype. I use this new memory block to subtract the size of the memory block from the variable holding the total memory use. Here I also need to include the size of the `alloc_t` datatype to remove the correct amount of memory. Lastly, I set the status of the memory block to 0. This marks the memory block as free for future use.

pfree:

The `pfree` function is used to free a block of page-aligned memory. Similarly to the `free` function this function takes a void pointer `mem` which is a pointer to the address of the page aligned memory block I want to free. The function starts by checking that the memory block is actually in the page-aligned heap by checking that it lies between `pheap_begin` and `pheap_end`. If it does not the program simply returns. Next, I get the memory address of the memory block and store it in the variable `ad`. I then use this variable to calculate the index of the memory block. This is done by subtracting the start address of the page heap from the memory block then dividing it by the page size (4096 bytes). I then use this index in the `pheap_desc` array to mark the memory block as free for future use.

pmalloc:

The `pmalloc` function is used to allocate a block of page-aligned memory. It starts by going into a for loop that loops through every possible page descriptor. If the current page descriptor is set to 1 the page is already in use and the `continue` statement makes the loop start from the top. If instead the page descriptor is set to 0 the page is free and I allocate it. This is done by setting the page descriptor for the page to 1 (in use), returning the address of the available page, and printing the address that is being allocated. If the entire for loop runs without finding a single available page the function prints an error message and returns.

malloc:

The `malloc` function is used to allocate a block of memory. It takes the parameter `size_t size` which corresponds to the size of memory needed. Firstly the function checks that the entered size is not 0. If it is the function simply returns. Then the function creates a pointer `mem` that points to the start of the heap. It then uses the address of the pointer `mem`, and the address of the last allocated memory block `last_alloc` to start a while loop. This loop will continue until the `mem` pointer reaches the address of `last_alloc`.

Inside the while loop:

A new `alloc_t` pointer `a` is created and set to point to the current memory block `mem`. If the size of the current memory block is 0 the function jumps to the `nalloc` label to allocate a new block at the end of the currently allocated memory. For any block with a size over 0 the while loop will continue. If the status of the current memory block is set to

allocated then the function skips over the current block by incrementing it with the size of the current block plus the size of the `alloc_t` datatype plus 4 bytes. After incrementing the current block the `continue` statement makes the while loop start again from the top. Finally if the size of the current block is equal to or larger than the requested size, and the block is free there has successfully been found a memory block that can be allocated. This memory block is then set to allocated, the allocated address space is printed, the total memory in use is incremented by the size of the block, the data in the memory block is cleared using `my_memset`, and the memory block is returned. Lastly if the memory block was too small the `mem` pointer gets incremented and the loop continues.

After the while loop:

If there is not found any memory blocks that are big enough and free before we reach the address of the last allocated memory the while loop is exited. Now the only way to allocate a memory block is to create a completely new memory block after the address of the last allocated memory block. First the function checks that there is enough memory between here and the end of the heap. If there is not the function prints an error and returns. Instead, if there is enough memory left the function creates a new memory block with the exact size needed and its status set to allocated. This is now the memory block closest to the end of the heap. Therefore, the `last_alloc` variable is updated to instead contain the address of the end of the new memory block. This is done by simply incrementing it with the size of the block plus the size of the `alloc_t` datatype plus 4 bytes. Lastly the function prints the allocated size and address space, increments the total memory used, sets the data in the memory block to 0 using `my_memset`, and returns the memory block.

my_memset

The `my_memset` function is a function that sets the value of a chosen number of consecutive bytes from a chosen address to a chosen variable. The function takes three parameters: a void pointer `str` that holds the chosen address, an int `len` that hold the number of consecutive bytes you wanna change, and an int `c` that holds the value you want the bytes changed into. Next, the function creates a char pointer `p` that points to the address stored in `str`. A while loop is then entered that loops `len` number of times. Every time the loop loops every next consecutive byte is then set to the value of `c`. Lastly, the function returns the original pointer `str`.

5.1.4 paging.c

```
1  #include "memory_management.h"
2
3  static uint32_t* page_directory = 0;
4  static uint32_t page_dir_loc = 0;
5  static uint32_t* last_page = 0;
6
7  void paging_map_virtual_to_phys(uint32_t virt, uint32_t phys)
8  {
9      uint16_t id = virt >> 22;
10     for(int i = 0; i < 1024; i++)
11     {
12         last_page[i] = phys | 3;
13         phys += 4096;
14     }
15     page_directory[id] = ((uint32_t)last_page) | 3;
16     last_page = (uint32_t *)(((uint32_t)last_page) + 4096);
17 }
18
19 void paging_enable()
20 {
21     asm volatile("mov %%eax, %%cr3": : "a"(page_dir_loc));
22     asm volatile("mov %cr0, %eax");
23     asm volatile("orl $0x80000000, %eax");
24     asm volatile("mov %eax, %cr0");
25 }
26
27 void init_paging()
28 {
29     printf("Setting up paging\n");
30     page_directory = (uint32_t*)0x400000;
31     page_dir_loc = (uint32_t)page_directory;
32     last_page = (uint32_t *)0x404000;
33     for(int i = 0; i < 1024; i++)
34     {
35         page_directory[i] = 0 | 2;
36     }
37     paging_map_virtual_to_phys(0, 0);
38     paging_map_virtual_to_phys(0x400000, 0x400000);
39     paging_enable();
40     printf("Paging was successfully enabled!\n");
41 }
42
```

The `paging.c` file starts by declaring three global pointer variables: `page_directory`, `page_dir_loc`, and `last_page` for memory management. `Page_directory` points to the start of the page directory. `Page_dir_loc` points to the physical location of the page directory. Lastly, `last_page` points to the last allocated page.

`paging_map_virtual_to_phys:`

The function `paging_map_virtual_to_phys` is used to map virtual addresses to physical addresses. It takes the virtual address and physical address as its parameters. The function starts by creating a variable `id` that holds the upper 10 bits of the virtual address. This value can then be used as an index in the page directory. Next, the function loops over all the 1024 entries in the page table. For each iteration it updates the current entry with the physical address, and sets the entry as both present and writeable. This is done for all pages from the given physical address with each subsequent page address being 4KB higher than the previous. Once the while loop has concluded all pages have been mapped. Following this the function makes the corresponding entry in the page directory point to this page table using the `id` variable. Lastly, the `last_page` pointer is updated to point to the next page in memory.

`paging_enable:`

The `paging_enable` function is used to enable paging in my operating system. This is done by moving the physical address of my page directory into the CR3 register. The CR3 register holds the physical base address of the page directory currently in use. Lastly, it enables paging by setting the paging bit of the CR0 register. The CR0 register holds various control flags that modify the basic operations of the processor.

`init_paging:`

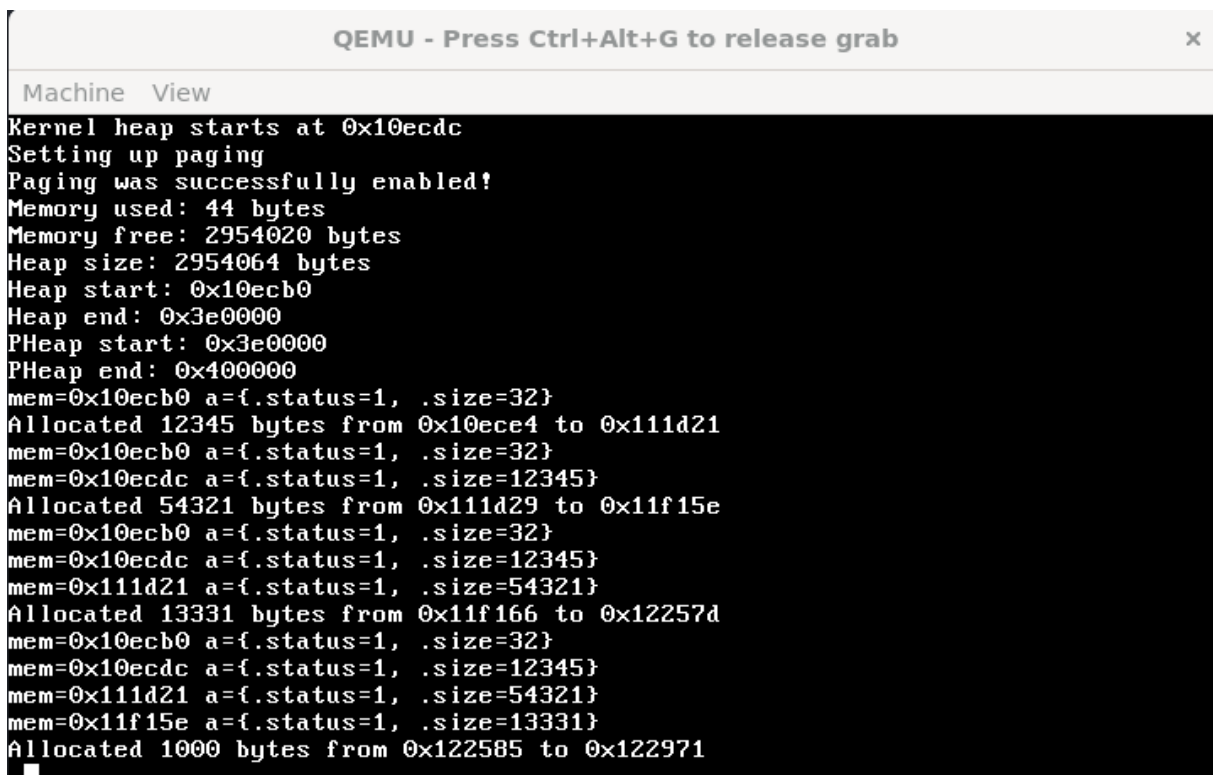
The `init_paging` function initializes the paging mechanism of my operating system. It starts by setting up the location in memory of the page directory and last page variables. Then it initializes every entry in the page directory as not present with supervisor-level read/write permissions. Next, the function maps the first 4MB of virtual memory to physical memory as well as the next 4MB of virtual memory to physical memory. Lastly, the function enables paging by calling the `paging_enable` function.

5.1.5 Functionality

I can now test if memory management is working in my operating system by allocating some memory using this code:

```
1 void* some_memory = malloc(12345);
2 void* memory2 = malloc(54321);
3 void* memory3 = malloc(13331);
4 char* memory4 = new char[1000]();
```

Which gives me this output:



The screenshot shows a terminal window titled "QEMU - Press Ctrl+Alt+G to release grab". The terminal output displays the following information:

```
Machine View
Kernel heap starts at 0x10ecdc
Setting up paging
Paging was successfully enabled!
Memory used: 44 bytes
Memory free: 2954020 bytes
Heap size: 2954064 bytes
Heap start: 0x10ecb0
Heap end: 0x3e0000
PHeap start: 0x3e0000
PHeap end: 0x400000
mem=0x10ecb0 a={.status=1, .size=32}
Allocated 12345 bytes from 0x10ece4 to 0x111d21
mem=0x10ecb0 a={.status=1, .size=32}
mem=0x10ecdc a={.status=1, .size=12345}
Allocated 54321 bytes from 0x111d29 to 0x11f15e
mem=0x10ecb0 a={.status=1, .size=32}
mem=0x10ecdc a={.status=1, .size=12345}
mem=0x111d21 a={.status=1, .size=54321}
Allocated 13331 bytes from 0x11f166 to 0x12257d
mem=0x10ecb0 a={.status=1, .size=32}
mem=0x10ecdc a={.status=1, .size=12345}
mem=0x111d21 a={.status=1, .size=54321}
mem=0x11f15e a={.status=1, .size=13331}
Allocated 1000 bytes from 0x122585 to 0x122971
```

Figure 12: Example of my memory management working

5.2 Programmable Interval Timer

5.2.1 Theory

The Programmable Interval Timer (PIT) is a crucial component in most computer systems [22]. It is a hardware chip that is connected to IRQ0 that can produce an interrupt signal at precise programmable intervals. These intervals are used by the operating system to

set up the system clock, schedule tasks, create time delays, and track time. The chip has an internal clock that oscillates at roughly 1.1931 MHz. This signal is then run through a frequency divider to produce the desired output frequency. The PIT is composed of three distinct channels each with its own frequency divider. Channel 0 is directly connected to IRQ0 and controls system clock functions. This is the channel I will be using for my operating system. Channel 1 controls refresh rates for DRAM, and channel 2 regulates the computer's speakers [22].

5.2.2 pit.h

My pit.h file is identical to the header file provided in assignment 4 [23]. The file is mostly used for definitions so there is no point in writing a custom one myself.

```
1  #ifndef TIMER_H
2  #define TIMER_H
3
4  #include <stdint.h>
5  #include <stdbool.h>
6
7  #define PIT_CMD_PORT 0x43
8  #define PIT_CHANNEL0_PORT 0x40
9  #define PIT_CHANNEL1_PORT 0x41
10 #define PIT_CHANNEL2_PORT 0x42
11 #define PC_SPEAKER_PORT 0x61
12 #define PIT_DEFAULT_DIVISOR 0x4E20
13
14 #define PIC1_CMD_PORT 0x20
15 #define PIC1_DATA_PORT 0x20
16 #define PIC_EOI 0x20
17
18 #define PIT_BASE_FREQUENCY 1193180
19 #define TARGET_FREQUENCY 1000
20 #define DIVIDER (PIT_BASE_FREQUENCY / TARGET_FREQUENCY)
21 #define TICKS_PER_MS (TARGET_FREQUENCY / TARGET_FREQUENCY)
22
23 void init_pit();
24 void sleep_interrupt(uint32_t milliseconds);
25 void sleep_busy(uint32_t milliseconds);
26
27 #endif
```

My `pit.h` file includes definitions for a Programmable Interval Timer and some functions related to it in the system. The first six definitions define the PIT-related macros. These include the port addresses for the pit channels, pc speaker, and cmd, as well as the value for the PIT default divisor. The next three definitions define the IRQ0-related macros. These include the port addresses for the command and data register of the master PIC and the value for the end-of-interrupt command code. The last four definitions define the custom sleep function constants for my PIT. These include the pit base frequency, target frequency, divider, and ticks/ms. Lastly the `pit.h` file declares the functions `init_pit`, `sleep_interrupt`, and `sleep_busy`.

5.2.3 `pit.cpp`

My `pit.cpp` file is based on the osdev tutorial by James molloy and is the implementation of my Programmable Interval Timer [22].

```
1  #include "pit.h"
2  #include "ports.h"
3  #include "isr.h"
4  #include "../drivers/screen.h"
5
6  uint32_t tick = 0;
7
8  static void timer_handler(registers_t regs)
9  {
10     tick++;
11
12     uint8_t cursor_end = inb(0x3D5);
13
14     outb(0x3D4, 0x0B);
15
16     outb(0x3D5, cursor_end ^ 0x80);
17 }
18
19 void init_pit()
20 {
21     register_interrupt_handler(IRQ0, timer_handler);
22
23     uint32_t divisor = DIVIDER;
24
25     outb(PIT_CMD_PORT, 0x36);
26
27     uint8_t low = (uint8_t)(divisor & 0xFF);
```

```

28     uint8_t high = (uint8_t)((divisor>>8) & 0xFF );
29
30     outb(PIT_CHANNEL0_PORT, low);
31     outb(PIT_CHANNEL0_PORT, high);
32 }
33
34 uint32_t get_current_tick()
35 {
36     return tick;
37 }
38
39 void sleep_interrupt(uint32_t milliseconds)
40 {
41     uint32_t current_tick = get_current_tick();
42     uint32_t ticks_to_wait = milliseconds * TICKS_PER_MS;
43     uint32_t end_ticks = current_tick + ticks_to_wait;
44
45     while (current_tick < end_ticks)
46     {
47         asm volatile(
48             "sti\n\t"
49             "hlt\n\t"
50         );
51         current_tick = get_current_tick();
52     }
53
54 }
55
56 void sleep_busy(uint32_t milliseconds)
57 {
58     uint32_t start_tick = get_current_tick();
59     uint32_t ticks_to_wait = milliseconds * TICKS_PER_MS;
60     uint32_t elapsed_ticks = 0;
61
62     while (elapsed_ticks < ticks_to_wait)
63     {
64         while(get_current_tick() == start_tick + elapsed_ticks)
65         {
66             //Do nothing (busy wait)
67         }
68         elapsed_ticks++;
69     }
70
71 }

```

The pit.cpp file starts by initializing a global variable tick that keeps track of the system

ticks.

timer_handler:

Then the `timer_handler` function is defined which is a handler function that will be set to be called by the IRQ0 handler. This function increments the tick count with each call and toggles the visibility of the cursor on and off by XORing the cursor end register with 0x80.

init_pit:

The next function `init_pit` is the function that initializes the PIT for my operating system. It starts by registering the `timer_handler` function as the handler for IRQ0. Then it creates a new variable `divisor` and sets it to the `DIVIDER` variable calculated in `pit.h`. This divisor variable is sent to channel 0 of the PIT in two parts: the low byte and the high byte.

get_current_tick:

The next part of the code is about creating the functionality for the operating system to sleep using interrupts or busy waiting. This implementation is based on the pseudo code provided in assignment 4 [23]. Both the functions for sleeping using interrupts, and using busy waiting need to know the system's current tick. Therefore, I started by creating a simple function called `get_current_tick` that returns the current tick.

sleep_interrupt:

The `sleep_interrupt` function takes the 32-bit integer milliseconds as its parameter. It starts by declaring three variables: `current_tick`, `ticks_to_wait`, and `end_ticks`. `Current_tick` holds the current tick, `ticks_to_wait` holds the number of ticks needed to wait for the entered milliseconds to pass, and `end_ticks` holds the number of ticks the system should be at after the wait time. Then the function enters a while loop that checks if the `current_tick` is less than the `end_ticks`. If it is not the CPU will be halted and interrupts will be enabled until the required number of ticks has passed. The CPU is then essentially put to sleep and wakes up for each tick to check whether it is time to exit the sleep state.

sleep_busy:

The `sleep_busy` shares a lot of similarities with `sleep_interrupt`. It has the same parameter and declares the same variables. The main difference is found inside the while loop. This function does not halt the CPU or enable interrupts. Instead, it continuously checks whether the required number of ticks has passed. This approach can be less effective as

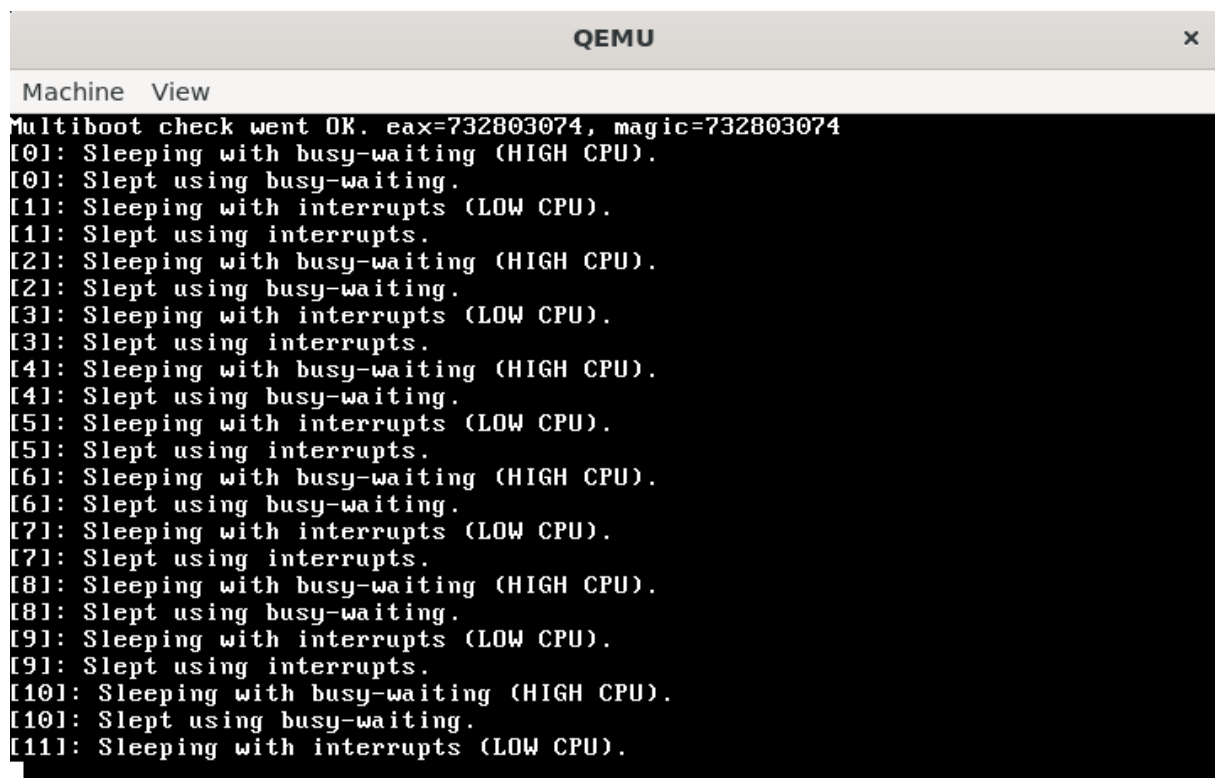
it keeps the CPU active during the entire sleep duration. Even so, it can still be useful in contexts where interrupts are disabled.

5.2.4 Functionality

I can check that my Programmable Interval Timer and sleep functions work by using this code in main.cpp [23]:

```
1  int counter = 0;
2  while(true)
3  {
4      printf("[%d]: Sleeping with busy-waiting (HIGH CPU).\n", counter);
5      sleep_busy(1000);
6      printf("[%d]: Slept using busy-waiting.\n", counter++);
7
8      printf("[%d]: Sleeping with interrupts (LOW CPU).\n", counter);
9      sleep_interrupt(1000);
10     printf("[%d]: Slept using interrupts.\n", counter++);
11 };
```

Which gives me this output:



```
Machine View
Multiboot check went OK. eax=732803074, magic=732803074
[0]: Sleeping with busy-waiting (HIGH CPU).
[0]: Slept using busy-waiting.
[1]: Sleeping with interrupts (LOW CPU).
[1]: Slept using interrupts.
[2]: Sleeping with busy-waiting (HIGH CPU).
[2]: Slept using busy-waiting.
[3]: Sleeping with interrupts (LOW CPU).
[3]: Slept using interrupts.
[4]: Sleeping with busy-waiting (HIGH CPU).
[4]: Slept using busy-waiting.
[5]: Sleeping with interrupts (LOW CPU).
[5]: Slept using interrupts.
[6]: Sleeping with busy-waiting (HIGH CPU).
[6]: Slept using busy-waiting.
[7]: Sleeping with interrupts (LOW CPU).
[7]: Slept using interrupts.
[8]: Sleeping with busy-waiting (HIGH CPU).
[8]: Slept using busy-waiting.
[9]: Sleeping with interrupts (LOW CPU).
[9]: Slept using interrupts.
[10]: Sleeping with busy-waiting (HIGH CPU).
[10]: Slept using busy-waiting.
[11]: Sleeping with interrupts (LOW CPU).
```

Figure 13: Sleeping using interrupts and busy waiting

5.3 Music player

I did do the optional music player task but I am not gonna provide the code used. This is because it is a substantial amount of code, none on which I wrote myself. For this task I followed the steps and used the code provided in assignment 4 [23]. I added the necessary lines of code to the tasks.json file and copied the apps directory to its needed destination. After running the subdirectory command in CMake my music player worked perfectly without any issues. I used this provided code to test my music player:

```
1 Song* songs[] = {
2     new Song(music_1, sizeof(music_1) / sizeof(Note)),
3     new Song(music_6, sizeof(music_6) / sizeof(Note)),
4     new Song(music_5, sizeof(music_5) / sizeof(Note)),
5     new Song(music_4, sizeof(music_4) / sizeof(Note)),
6     new Song(music_3, sizeof(music_3) / sizeof(Note)),
7     new Song(music_2, sizeof(music_2) / sizeof(Note))
8 };
9 uint32_t n_songs = sizeof(songs) / sizeof(Song*);
10
11 SongPlayer* player = create_song_player();
12
13
14 while(true){
15     for(uint32_t i =0; i < n_songs; i++){
16         printf("Playing Song...\n");
17         player->play_song(songs[i]);
18         printf("Finished playing the song.\n");
19     }
20 };
```

Which played music and gave me this output:

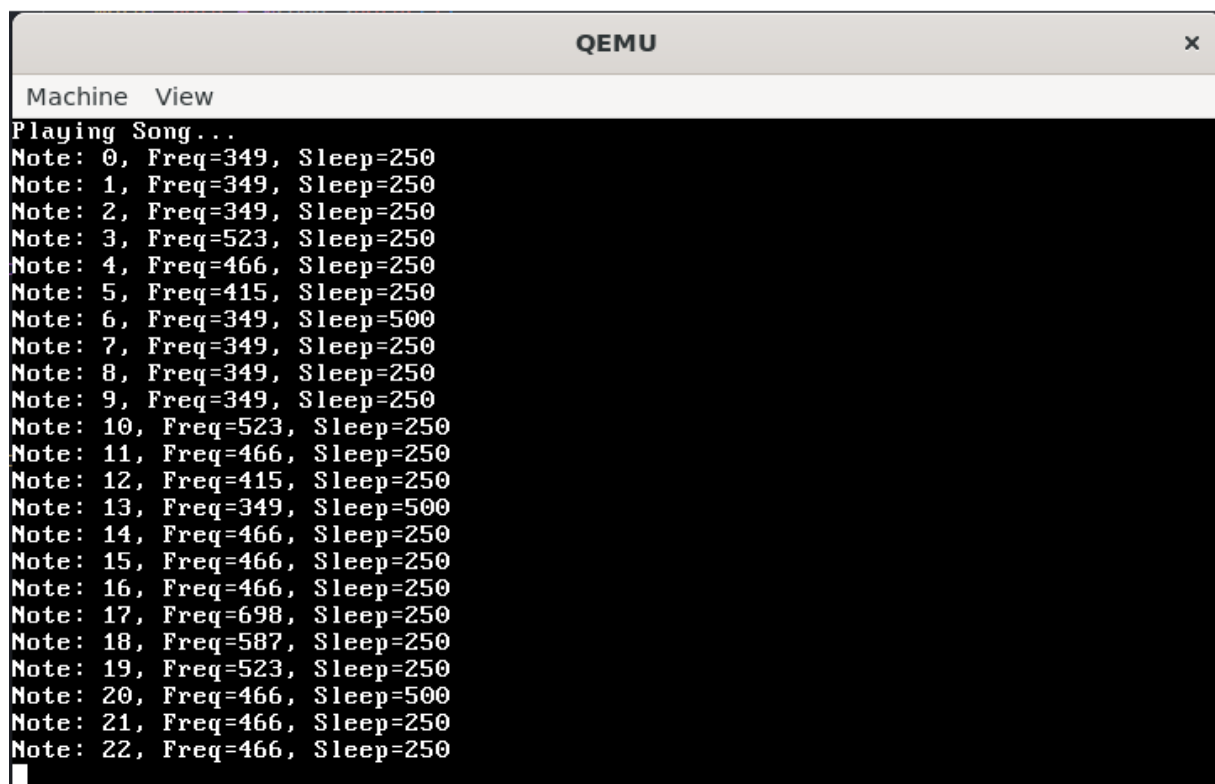





Figure 14: Music player in action

5.4 Successful Pull Request to Main Repo


UIA OSDev. Siverteh, group 37 #35











 **Open** Siverteh wants to merge 10 commits into `uiaict:master` from `Siverteh:master` 


Conversation 0 Commits 10 Checks 0 Files changed 56

 Siverteh commented 14 hours ago ...

No description provided.

 Siverteh added 10 commits 4 months ago

-  Finished setting up environment d6208a3
-  Added assignment 2 code f4d2e33
-  Added assignment 2 code 2cc61e8
-  Added idt, interrupts, and keyboard functionality b414a4d
-  Added blinking cursor functionality 4c74543
-  Commented parts of the code b997468
-  Finished my operating system c2f13aa
-  Final operating system 864a7be
-  Finished my operating system 17c709d
-  Finished my operating system 0006f3e

 Siverteh changed the title ~~My final operating system: Siverteh, gruppe 37~~ UIA OSDev. Siverteh, group 37 now

Add more commits by pushing to the `master` branch on `Siverteh/ikt218-osdev`.



  **This branch has no conflicts with the base branch**
Only those with [write access](#) to this repository can merge pull requests.

Figure 15: Successful pull request to main repo

5.5 Approach and challenges

I had a lot more trouble finishing this assignment compared to the previous ones. The task I struggled with was implementing paging. I tried to follow James Molloy's guide on paging but I was unable to get the code to work for my operating system [24]. After giving up on this code I tried various other paging guides to no avail. Every implementation I had on paging got my operating system stuck in a boot loop. I tried to follow three different guides on paging, but every single guide made my system end up in a boot

loop. Eventually I gave up on my own implementation of paging and resorted to using the paging code provided in assignment 4's assignment files. My approach then changed to understanding this code and being able to explain it well. I got this code to work well in my operating system, and I got a good understanding of how it worked. My implementation of the PIT was super simple as I had already implemented it beforehand. I wanted to have a blinking mouse cursor in my terminal when I was creating my keyboard handler and needed the PIT for this. Implementing the PIT as fairly simple in general.

6 Conclusion

The object of this project was to learn about operating system development by developing my own operating system from scratch. To accomplish this I attended most lectures and read various sources online to gain knowledge on the subject. I also spent a lot of time playing around with my development environment. After finishing my project I feel like I have succeeded at this task as I have learned a substantial amount about operating systems. I am happy with the results of my operating system as I was able to implement every task in a way that felt satisfactory. Furthermore, I have gotten a deeper appreciation for all the nuts and bolts that goes into creating a functional operating system. In summary this project has succeeded in its learning outcomes and I am happy with the overall result.

Bibliography:

- [1] T. Fisher. “What is post?” (), [Online]. Available: <https://www.lifewire.com/what-is-post-2625953>. (accessed: 01.02.2023).
- [2] Wikipedia, the free encyclopedia, *Power-on self-test*, [Online; accessed February 1, 2023], 2023. [Online]. Available: https://en.wikipedia.org/wiki/Power-on_self-test#/media/File:POST_P5KPL.jpg.
- [3] G. Duarte. “How computers boot up.” (), [Online]. Available: <https://manybutfinite.com/post/how-computers-boot-up/>. (accessed: 01.02.2023).
- [4] E. Lumunge. “Types of boot loaders.” (), [Online]. Available: <https://iq.opengenus.org/types-of-bootloaders/>. (accessed: 01.02.2023).
- [5] E. Seattle. “Overview of boot options in windows.” (), [Online]. Available: <https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/boot-options-in-windows>. (accessed: 24.02.2023).
- [6] F. Rosner. “Writing my own boot loader.” (), [Online]. Available: <https://dev.to/frosnerd/writing-my-own-boot-loader-3mld>. (accessed: 01.02.2023).
- [7] P. Anvin. “The linux/i386 boot protocol.” (), [Online]. Available: http://kerneltravel.net/blog/1/zzp_report4/. (accessed: 02.02.2023).
- [8] Unknown, *Memory map (x86)*, [Online; accessed February 4, 2023], 2020. [Online]. Available: [https://wiki.osdev.org/Memory_Map_\(x86\)](https://wiki.osdev.org/Memory_Map_(x86)).
- [9] P-A. Andersen. “Lecture 2 - x86 and the bootloader.” (), [Online]. Available: <https://perara.notion.site/Lecture-2-x86-and-the-bootloader-d9c08d86e6c7442fa939861506a84b2a>. (accessed: 09.03.2023).
- [10] J. Molloy. “Genesis.” (), [Online]. Available: http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html. (accessed: 09.03.2023).
- [11] Unknown. “Bare bones with nasm.” (), [Online]. Available: https://wiki.osdev.org/Bare_Bones_with_NASM. (accessed: 20.03.2023).
- [12] J. Molloy. “The gdt and idt.” (), [Online]. Available: http://www.jamesmolloy.co.uk/tutorial_html/4.-The%5C%20GDT%5C%20and%5C%20IDT.html. (accessed: 06.03.2023).
- [13] Unknown. “Operating system development protected mode and the global descriptor table (gdt).” (), [Online]. Available: <https://www.independent-software.com/operating-system-development-protected-mode-global-descriptor-table.html>. (accessed: 03.04.2023).
- [14] HandWiki. “Vga-compatible text mode.” (), [Online]. Available: https://handwiki.org/wiki/VGA-compatible_text_mode. (accessed: 01.06.2023).
- [15] 25dikshasinghal. “Interrupts and exceptions.” (), [Online]. Available: <https://www.geeksforgeeks.org/interrupts-and-exceptions/>. (accessed: 20.04.2023).

- [16] Unknown. "Interrupt descriptor table." (), [Online]. Available: https://wiki.osdev.org/Interrupt_Descriptor_Table. (accessed: 23.04.2023).
- [17] Unknown. "Idt descriptors." (), [Online]. Available: https://pdos.csail.mit.edu/6.828/2008/readings/i386/s09_05.htm. (accessed: 13.04.2023).
- [18] V. Palaniveloo. "Interrupt service routine." (), [Online]. Available: <https://medium.com/@vinita.palaniveloo/interrupt-service-routine-eb38e5dd5058>. (accessed: 20.04.2023).
- [19] R. Awati. "Interrupt request (irq)." (), [Online]. Available: <https://www.techtarget.com/whatis/definition/IRQ-interrupt-request>. (accessed: 20.04.2023).
- [20] J. Molloy. "5. irqs and the pit." (), [Online]. Available: http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%5C%20and%5C%20the%5C%20PIT.html. (accessed: 05.05.2023).
- [21] T. Contributor. "Paging." (), [Online]. Available: <https://www.techtarget.com/whatis/definition/paging>. (accessed: 01.06.2023).
- [22] J. Molloy. "5. irqs and the pit." (), [Online]. Available: http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%5C%20and%5C%20the%5C%20PIT.html. (accessed: 25.05.2023).
- [23] P.-A. Andersen. "Assignment 4 - memory and pit." (), [Online]. Available: <https://perara.notion.site/Assignment-4-Memory-and-PIT-2dbf775240da488299c67828f5ce8e93>. (accessed: 01.06.2023).
- [24] J. Molloy. "6. paging." (), [Online]. Available: http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html. (accessed: 05.05.2023).

Online sources

- [1] T. Fisher. "What is post?" (), [Online]. Available: <https://www.lifewire.com/what-is-post-2625953>. (accessed: 01.02.2023).
- [3] G. Duarte. "How computers boot up." (), [Online]. Available: <https://manybutfinite.com/post/how-computers-boot-up/>. (accessed: 01.02.2023).
- [4] E. Lumunge. "Types of boot loaders." (), [Online]. Available: <https://iq.opengenus.org/types-of-bootloaders/>. (accessed: 01.02.2023).
- [5] E. Seattle. "Overview of boot options in windows." (), [Online]. Available: <https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/boot-options-in-windows>. (accessed: 24.02.2023).
- [6] F. Rosner. "Writing my own boot loader." (), [Online]. Available: <https://dev.to/frosnerd/writing-my-own-boot-loader-3mld>. (accessed: 01.02.2023).
- [7] P. Anvin. "The linux/i386 boot protocol." (), [Online]. Available: http://kerneltravel.net/blog/1/zzp_report4/. (accessed: 02.02.2023).

- [9] P.-A. Andersen. "Lecture 2 - x86 and the bootloader." (), [Online]. Available: <https://perara.notion.site/Lecture-2-x86-and-the-bootloader-d9c08d86e6c7442fa939861506a84b2a>. (accessed: 09.03.2023).
- [10] J. Molloy. "Genesis." (), [Online]. Available: http://www.jamesmolloy.co.uk/tutorial_html/2.-Genesis.html. (accessed: 09.03.2023).
- [11] Unknown. "Bare bones with nasm." (), [Online]. Available: https://wiki.osdev.org/Bare_Bones_with_NASM. (accessed: 20.03.2023).
- [12] J. Molloy. "The gdt and idt." (), [Online]. Available: http://www.jamesmolloy.co.uk/tutorial_html/4.-The%5C%20GDT%5C%20and%5C%20IDT.html. (accessed: 06.03.2023).
- [13] Unknown. "Operating system development protected mode and the global descriptor table (gdt)." (), [Online]. Available: <https://www.independent-software.com/operating-system-development-protected-mode-global-descriptor-table.html>. (accessed: 03.04.2023).
- [14] HandWiki. "Vga-compatible text mode." (), [Online]. Available: https://handwiki.org/wiki/VGA-compatible_text_mode. (accessed: 01.06.2023).
- [15] 25dikshasinghal. "Interrupts and exceptions." (), [Online]. Available: <https://www.geeksforgeeks.org/interrupts-and-exceptions/>. (accessed: 20.04.2023).
- [16] Unknown. "Interrupt descriptor table." (), [Online]. Available: https://wiki.osdev.org/Interrupt_Descriptor_Table. (accessed: 23.04.2023).
- [17] Unknown. "Idt descriptors." (), [Online]. Available: https://pdos.csail.mit.edu/6.828/2008/readings/i386/s09_05.htm. (accessed: 13.04.2023).
- [18] V. Palaniveloo. "Interrupt service routine." (), [Online]. Available: <https://medium.com/@vinita.palaniveloo/interrupt-service-routine-eb38e5dd5058>. (accessed: 20.04.2023).
- [19] R. Awati. "Interrupt request (irq)." (), [Online]. Available: <https://www.techtarget.com/whatis/definition/IRQ-interrupt-request>. (accessed: 20.04.2023).
- [20] J. Molloy. "5. irqs and the pit." (), [Online]. Available: http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%5C%20and%5C%20the%5C%20PIT.html. (accessed: 05.05.2023).
- [21] T. Contributor. "Paging." (), [Online]. Available: <https://www.techtarget.com/whatis/definition/paging>. (accessed: 01.06.2023).
- [22] J. Molloy. "5. irqs and the pit." (), [Online]. Available: http://www.jamesmolloy.co.uk/tutorial_html/5.-IRQs%5C%20and%5C%20the%5C%20PIT.html. (accessed: 25.05.2023).
- [23] P.-A. Andersen. "Assignment 4 - memory and pit." (), [Online]. Available: <https://perara.notion.site/Assignment-4-Memory-and-PIT-2dbf775240da488299c67828f5ce8e93>. (accessed: 01.06.2023).
- [24] J. Molloy. "6. paging." (), [Online]. Available: http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html. (accessed: 05.05.2023).

Picture sources

- [2] Wikipedia, the free encyclopedia, *Power-on self-test*, [Online; accessed February 1, 2023], 2023. [Online]. Available: https://en.wikipedia.org/wiki/Power-on_self-test#/media/File:POST_P5KPL.jpg.
- [8] Unknown, *Memory map (x86)*, [Online; accessed February 4, 2023], 2020. [Online]. Available: [https://wiki.osdev.org/Memory_Map_\(x86\)](https://wiki.osdev.org/Memory_Map_(x86)).