Operating System

Sam Pieters

April 2022

Contents

Part I
Planning & Milestones

1 Motivation

For the subject 'individual project' I want to write an operating system under the name 'SammyOS'. The languages I am going to use are x86 assembly (x86-64) and C. I am going to start as soon as possible (so no exact beginning date), this is the reason why I will express all my goals I want to make in weeks. I will give an update every time a milestone is reached, the exact end date and every deviation (things I did not mention in this report) together with the sources I used to complete the specific part. There will probably be many deviations from the point this report is made, this is because I only know the basics of how an operating system works. I chose this project to have a knowledge of what lies behind the logic in a computer and how it's all connected.

2 Schedule

Part	necessity	Week
Setup	required	1
BIOS and bootloader	required	1
Simple kernel	required	1
Simple display driver (VGA)	required	1
Keyboard driver	required	1
Physical memory	required	2
Virtual memory	required	2
Processes	required	2
File system	required	3
Kernel/Shell commands	optional	1
GUI	optional	2
Mouse driver	optional	1

3 Accessories

To run the operating system on a virtual environment, the virtual environment needs to have the same accessories enabled as explained in part II Installation & Setup. Then we need a computer which needs to satisfy the following:

- Highly recommended to use a computer without a hard disk or other peripherals.
- The processor needs to support 1G huge page.
- The computer has the possibility to connect a PS/2 keyboard and mouse.
- The processor needs to support 64-bit long mode.

In summary, the processor should be manufactured in the year 2013 or later and the computer has no other devices except the CPU and RAM including a way to connect the PS/2 devices.

Part II
Installation & Setup

Before I start with the operating system, a virtual machine is required to run the operating system. This is because a real machine would restart (over and over) if there are no interrupt handlers when exceptions occur. After it is tested in a virtual environment, and everything goes right, it can be tested in a real environment.

As discussed in the previous part (Part I Installation & Setup), bochs emulator is used because we can manually add some settings to our virtual environment (setting up the A20 line, ensuring long mode, memory size, ...) and can report error messages if the code causes exceptions.

4 Bochs

4.1 Setting up hard disk with Bochs

1. Run the command "bximage" in the command prompt to create a disk image file (if this doesn't work check if the path C:\Program Files\Bochs-2.6.11 is present win the system variables).
2. Enter the first option, Create new floppy or hard disk image. The rest of the options are of no use to the project right now.
3. Then the question is asked to create a floppy disk or hard disk. The hard disk is chosen because hard disk are commonly used nowadays.
4. Create a flat image. A flat image is one file with a flat layout. This means that all the sectors of the hard disk are stored in one flat file, in LBA order. LBA provides a simple linear address space to the host. The host only needs to provide the LBA address without knowing anything of the physical sector positions. Blocks are located by an index, with the first block being LBA=0, the second LBA=1, and so on. Flat image is chosen because it is the default and seems the easiest to program with.
5. The size of the hard disk sectors is not so important so it will be kept to 512 bytes (default) and change it later if needed.
6. The size of the disk is also not important for now so it will be kept at 10 megabytes.
7. We'll call the first image file "boot.img".
8. Now the image file is created and there are three values CHS=20/16/63 generated. CHS stands for cylinder-head-sector scheme, where blocks are addressed by means of a tuple which define the cylinder, head, and sector at which they appear on the hard disk. Because we use LBA order, CHS addresses can be converted to LBA addresses using the following formula:

   $$LBA = ((C * HPC) + H) * SPT + S - 1$$

   Where C, H and S are the cylinder number, the head number, and the sector number. LBA is the logical block address, HPC is the number of heads per cylinder and SPT is the number of sectors per track.

4.2 Setting up the virtual environment with Bochs

Now the Bochs app can be opened where some settings need to be changed in order to configure the virtual environment for our operating system environment. The following changes are briefly explained:

1. Go to “CPUID” and press “Edit”. Enable “x86-64 and long mode”, this is because we are going to built our kernel in long mode and write our lower level code in the x86-64 language.
2. Go to “CPUID” and press “Edit”. Enable “1G pages support in long mode”, This is enabled for the future when the operating system supports pages. This allows to address 1 gigabyte chunks of memory at once, lowering a bottleneck from processing a lot of pages.
3. Go to “Memory” and press “Edit”. The memory size and host allocated memory size is changed to 1024 MB. This is so that there is enough room when the project is progressing.
4. Go to “Disk & Boot” and press “Edit”. Go to “ATA channel 0” and press the subtab “First HD/CD on channel 0”. The path or physical device name is changed to “boot.img” because that is the name of the hard disk image.
5. Next, in the same subtab, the number we got in the last step of creating the environment (CHS=20/16/63) is added. This means there are 20 cylinders, 16 heads and 63 sectors per track. Fill in these numbers in the corresponding places.
6. Go to “Disk & Boot” and press “Edit”. Go to “Boot options” and change Boot drive #1 to disk.
7. Save all the options to the configuration file named “bochsrc”.

Now the boot image will be loaded as a disk device and we can select this device as the first device to boot. Now that this milestone is completed, assembly code is written to create the boot loader and to setup the kernel (next milestone). Be aware that most of the settings are setup later in the project but they are referenced here so that there is a good overview of the bochsrc file.

To run the bochsrc configuration file, the bochs app can be opened. The configuration file has to be loaded in the bochs app with the load button (and specify the path to the file). After that, start is pressed and the virtual environment will be run. Another way of running the environment is with the command:

```
bochs -f bochsrc.bxrc -q
```

The ”-f” option specifies the path to the configuration file and the option ”-q” tells bochs to quick start the environment (without displaying the configuration window first”.

5 The build script

There will be multiple types of files in the project (.asm files, .h files and .c files). All these files have to be converted to binary files so that they can be read from the disk image because the disk can only contain binary. The commands in the following subsections are imported in the ”build.sh” scripts. To run these scripts, use the following command:

```
sh ./build.sh
```

Just run the script and open the “bochsrc” file to check everything in the virtual environment.

5.1 Converting assembly files to binary files

Assembly files (.asm) are files that are written in the x86-64 language. These have to be converted to binary files. To do that, the netwide assembler (NASM) is used which is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit and 64-bit (x86-64) programs. It is considered one of the most popular assemblers for Linux. The following command line is used to convert an assembly file to a binary file:

```
nasm -f bin boot.asm -o boot.bin
```

The "-f" option is the file that needs to be converted, the "bin" statement is the format the file needs to be converted to and the "-o" option specifies the name of the output file.

5.2 Converting C files to binary files

To convert a C file (.c) to a binary file, the GNU Compiler Collection (GCC) is used. This is a collection of compilers. It's important to know that the program gcc itself is not a compiler but takes care that the right parts of the GCC compiler are called. The following command will convert a C file. Notice that the c99 standard is used, a past version of the C programming language standard. It extends the previous version (C90) with new features for the language and the standard library, and helps implementations make better use of available computer hardware, such as IEEE 754-1985 floating-point arithmetic, and compiler technology.

```
gcc -std=c99 -mcmodel=large -ffreestanding -fno-stack-protector
-mno-red-zone -c file.c
```

The "mcmodel=large" option is to ensure that a large code model is used. This allows no restrictions on code or data, accesses to both code and data use absolute addressing Instruction Pointer (IP)-relative addressing requires only 32 bits, whereas absolute addressing requires 64-bits. This can affect code size and performance (IP-relative addressing is somewhat faster.). Otherwise the system could have relocation truncated to fit error. The "ffreestanding" option sets a freestanding environment. A freestanding environment is one in which the standard library may not exist, and program startup may not necessarily be at "main". The option -ffreestanding" directs the compiler to not assume that standard functions have their usual definition. The "fno-stack-protector" option disables stack protection. With the stack protection there will be a little more space allocated on the stack and a little more overhead on entry to and return from a function while the code sets up the checks and then actually checks whether you've overwritten the stack while in the function.

The executable file that will be generated with the gcc command is an elf file. ELF stands for Executable and Linkable Format and is a common standard file format for executable files, object code, shared libraries, and core dumps. Each ELF file is made up of one ELF header, followed by file data. The data can include

- Program header table: describing zero or more memory segments.
- Section header table: describing zero or more sections.
- Data referred to by entries in the program header table or section header table.

The nasm assembler can output an elf file. The nasm command takes an x86-64 assembly file and generates an object file in the elf 64 bit format with the following command:

```
nasm -f elf64 file.asm -o file.o
```

After the object files are generated, they will have to be linked with other files written in C. To do that, ELF is used. ELF is a format for storing programs or fragments of programs on disk, created as a result of compiling and linking. An ELF file is divided into sections. For an executable program, these are the text section for the code, the data section for global variables and the rodata section that usually contains constant strings. These sections are also defined in the "link.lds" file. The ELF header contains all of the relevant information required to load an ELF executable. The format of this header is described in the ELF specification. The ELF file contains headers that describe how these sections should be stored in memory.

In the "kernel.asm" file, the section on top is removed and changed with sections (data and text).

These sections are necessary in an ELF file. The data section is where global tables, variables, etc. live and the text section is where the codes lives. Then we will enter kernel entry and jump to kernel main function in C. Before we call the main function, we adjust the kernel stack pointer (to address 0x200000) which we also use in C code. As last, before the C file is created, the start of the kernel needs to be globally declared so that the linker will find it. To call an external function, you need to tell the assembler the function is "extern". "extern" isn't actually an instruction and doesn't show up in the disassembly, it's just a message to the assembler, often called a pseudoinstruction.

5.3 Writing binary files in the boot image

Now the binary file has to be written in the boot image, this can be done with the dd command:

```
dd if=boot.bin bs=X count=Y seek=Z of=boot.img conv=notrunc
```

The input file is specified with the "if" option. The "bs" option tells us what the block size in bytes has to be. The option "count" means how many blocks we want to write. The "seek" option seeks the record specified by the Z variable from the beginning of output file before copying. At last we add "notrunc" because this option instructs dd to not truncate the output file. "noerror" is not added because if the command fails we know that something went wrong.

Part III
Bootloader

Before starting with the implementation of the bootloader, it is assumed that the BIOS is present and the BIOS services are available to the system. BIOS stands for Basic Input Output System, it initializes hardware and provides runtime services for an operating system. When a computer is booted, BIOS initializes and tests the hardware. This is called the Power-On Self-Test (or POST). If this procedure succeeds then the BIOS will find the boot device by the boot signature. The boot signature is a boot sector (sector number 0) and it contains the byte sequence 0x55, 0xAA at byte offsets 510 and 511 respectively. When the BIOS finds such a boot sector, it is loaded into memory at 0x0000:0x7c00 (segment 0, address 0x7c00). Execution is then transferred to the freshly loaded boot record. This boot record is often referred to as the master boot record (MBR).

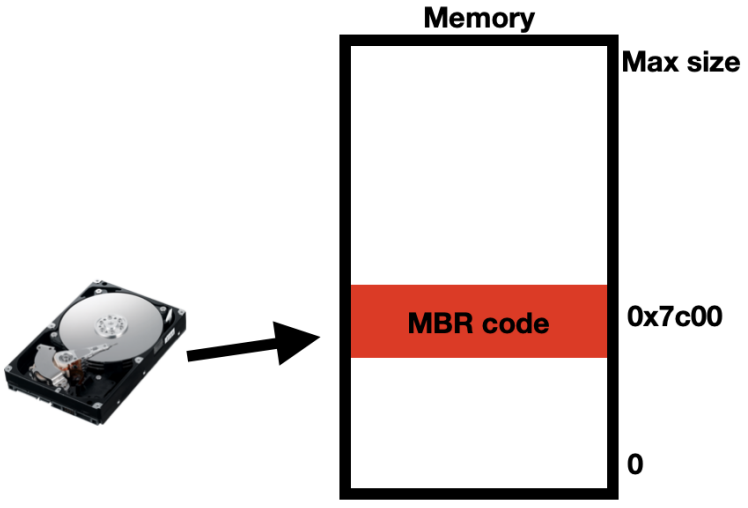

Figure 1: MBR in memory

6 Master Boot Record

The information on the first sector is also called the master boot record (MBR). The MBR has a total of 512 bytes and can' be any longer. From these 512 bytes, 2 bytes are used for the boot signature (0xAA55), 64 bytes are used for the partition table which is a table that stores entries to other files. The rest of the bytes are used as area for code, which is 446 bytes.

When booting the operating system, it will go through 3 different modes: real mode (16 bit mode), protected mode (32 bit mode) and long mode. Long mode consists of two sub modes which are 64 bit mode and compatibility mode. In compatibility mode, 16-bit or 32-bit programs can run under a 64-bit operating system without recompiling those programs. Because this operating system is going to be designed for 64-bit programs, we will focus only on the 64 bit mode and not compatibility mode.

In real mode, the only thing that must be performed is loading the kernel in memory and getting information about the hardware so that it can be used in the system. After the operating system will be running in long mode. There will be some functions needed to print messages on screen as soon as possible. This is necessary because when something goes wrong in the operating system, the error can be debugged using the messages on screen. The functions needed in real mode will be printing characters, disk services, memory map and video mode.

The MBR will setup the stack, which grows downwards, test disk extension and handle if something goes wrong in this part. The MBR is a fixed 512 bytes long and can't be longer.

6.1 Memory addressing in real mode

Before the system is coded we have to learn how memory addressing works in real mode. First off, there are 8 general purpose registers in 16-bit:

- AX: arithmetic operations
- BX: base index register for MOVE, points to data
- CX: shift/rotate instructions and loops
- DX: port address for IO operations and used for arithmetic operations
- SP: points to top of the stack
- BP: points to base of the stack
- SI: points to a source in stream operations
- DI: points to a destination in stream operations

These first 4 registers (ax, bx, cx and dx) will be used in the project in real mode. Note that 8 bit, 16 bit and also 32 bit registers can be used in real mode but 64 bit registers are not available in real mode.

More important are the 4 registers called segment registers, these have the size of 16 bit:

- CS: points to code (code segment), machine instructions exist at some offset into a code segment. The segment address of the code segment of the currently executing instruction is contained in CS.
- DS: points to data (data segment), variables and other data exist at some offset into a data segment. There may be many data segments, but the CPU may only use one at a time, by placing the segment address of that segment in register DS.
- SS: points to stack (stack segment), the stack is a very important component of the CPU used for temporary storage of data and addresses. Therefore, the stack has a segment address, which is contained in register SS.

- ES: points to extra data (extra segment), a spare segment that may be used for specifying a location in memory.

The default segment register for accessing data is the ds register.The format of the logical address is:

Logical Address = segment register:Offset

The format of the physical address is:

Physical Address = Segment * 16 + Offset

The prefix 0x or suffix h is used to represent a hexadecimal number. So this instruction will copy the data in the address at 0x10050 to register ax. An example of this is the following instruction: mov ax, [ds:50h] with value 1000 in register ds (ds = 0x1000). Notice that the segment register doesn't need to be specified thus instead of [ds:50h] we can also write [50h] because if no register is specified, the default register ds is used. This instruction will read the value in the address at ds:50 to ax. The memory address we get is 0x1000 * 16 + 0x50 = 0x10050.

6.2 Memory addressing in protected mode

In 32 bit mode, or protected mode, the 8 general purpose registers can also be used:

- EAX: arithmetic operations
- EBX: base index register for MOVE, points to data
- ECX: shift/rotate instructions and loops
- EDX: port address for IO operations and used for arithmetic operations
- ESP: points to top of the stack
- EBP: points to base of the stack
- ESI: points to a source in stream operations
- EDI: points to a destination in stream operations

The prefix "E" is added which stands for "extended" to indicate that it is a 32 bit register. Note that if the lower half of these registers is taken, it becomes the 16 bit version of these general purpose registers. An example of this is the 32 bit register EAX which becomes the 16 bit ax when divided in half.

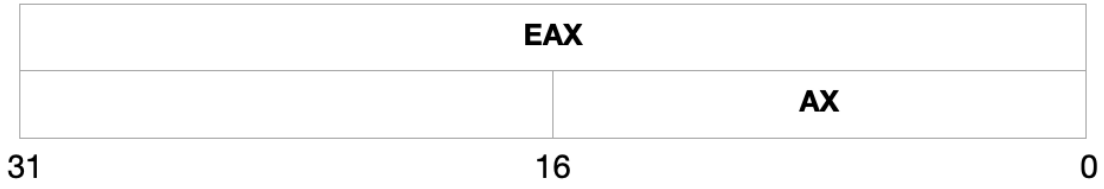

Figure 2: 32 bit register setup

The first 4 registers (EAX, EBX, ECX and EDX) will be the most used.

6.3 Memory addressing in long mode

In 64 bit mode, or long mode, the 8 general purpose registers can also be used:

- RAX: arithmetic operations
- RBX: base index register for MOVE, points to data
- RCX: shift/rotate instructions and loops
- RDX: port address for IO operations and used for arithmetic operations
- RSP: points to top of the stack
- RBP: points to base of the stack
- RSI: points to a source in stream operations
- RDI: points to a destination in stream operations

The prefix "R" is added which stands for "register" to indicate that it is a 64 bit register. Note that if the lower half of these registers is taken, it becomes the 32 bit version of these general purpose registers. An example of this is the 64 bit register RAX which becomes the 32 bit EAX when divided in half, this in it's turn becomes the 16 bit register AX when divided in half again.

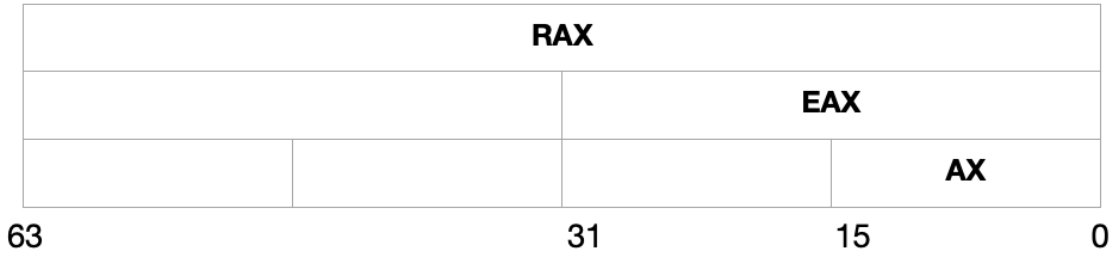

Figure 3: 64 bit register setup

The first 4 registers (RAX, RBX, RCX and RDX) will be the most used.

7 Testing disk extension services

Testing disk extension services is next. The BIOS provides a set of disk access routines using the int 0x13 family of BIOS functions. A small part of this family are the disk extension services. This will be used for physical sector-based disk read or write operations. The kernel needs to be loaded from the disk to the memory, when we are going to read more than one sector we need to provide the CHS value that was given in Part II Installation & Setup. The BIOS interrupt call for this is 13h. The CHS value is needed in order to locate the sector that needs to be read, because we are working in LBA order, the CHS value is converted to LBA with the formula explained in Part II Installation & Setup under the section "Setting up hard disk with Bochs. Modern computers support the disk extension services but it needs to be tested anyway. To check this we need value AH=41h, load 55AAh in register BX and the drive id (drive index) in register DL.

Part IV
Protected Mode

To load the kernel, the operating system needs to switch to long mode or 64 bit. To do this, several steps need to be taken including going to protected mode first. Every step before switching to long mode is explained in this part.

8 Load the loader

When testing disk extension services has passed, new files can be loaded from the disk into memory such as the new file that's made called "loader.asm".

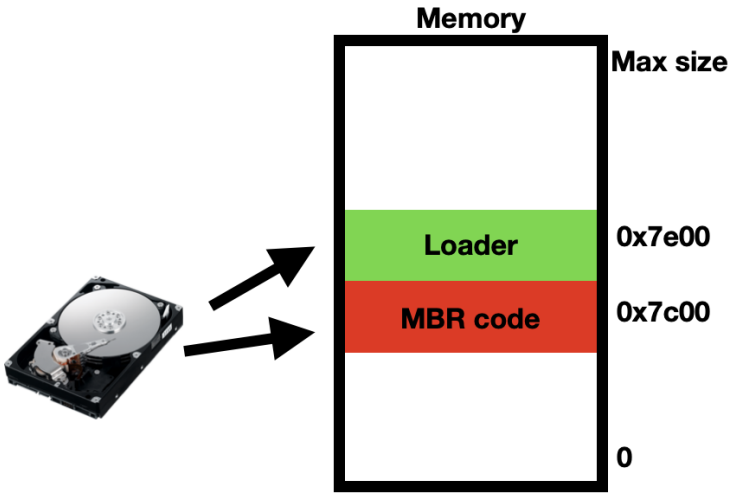

Figure 4: Loader in memory

A loader is needed because the MBR is 512 bytes long and every task that needs to be done in the loader is longer than 512 bytes in total. The loader file has no 512 bytes limit. The loader will load the kernel file in main memory, get memory map, switch to protected mode and long mode. The MBR starts on the address 0x7c00 and is 512 bytes long or 200 in hexadecimal. The loader can thus be placed on address 0x7c00 + 0x200 = 0x7e00 right behind the MBR.

To load the loader file, in "boot.asm" the function code 42 is saved in the ah register and the BIOS interrupt 0x13 is called. This means the disk extension service is used that was tested in the previous part (Part III Bootloader). This is going to read the loader file into memory. The next thing to do is jumping to the start of the loader. The loader file needs to be written, this is done in the "loader.asm" file. To successfully let multiple files work we need to convert this to a binary file (like "boot.asm" in the shell script) and write it into the boot image in the shell script.

8.1 Check dependencies

Next, the loader has to convert to long mode. So first off, the system checks if long mode is supported. To check this, the instruction "cpuid" from the x86-64 is needed. So we test if this instruction is available to the operating system. Bit 21 in the EFLAGS register (also called id flag) indicates support for the CPUID instruction. The instruction "cpuid" can be used if the check passes.

Now the instruction "cpupid" can be used to check whether long mode can be used. This instruction doesn't take a parameter and implicitly uses the EAX register as argument. To test if long mode is available, we need to call cpuid with 0x80000001 in EAX. This loads some information to the ECX and EDX registers. Long mode is supported if the 29th bit in EDX is set. We call the "cpupid" instruction two times. The reason is that the "cpuid" command started with only a few functions and was extended over time. So old processors may not know the 0x80000001 argument at all. To test if they do, we need to invoke cpuid with 0x80000000 in EAX first. It returns the highest supported parameter value in EAX. If it's at least 0x80000001, we can test for long mode as described above. Else the CPU is old and doesn't know what long mode is either.

1G page support is also checked which is at bit 26, thus if the 26th bit in edx is set then it is

supported. It is thus tested the same way as the check for long mode. Since the process address space are virtual, the CPU and the operating system have to remember which page belong to which process, and where it is stored. Obviously, the more pages there are, the more time it takes to find where the memory is mapped.

9 Setting up the kernel

After long mode check is passed, the kernel file can be loaded. This can be done in the same way as the loader was loaded in the boot file except some parameters have to change. First is decided at which memory space to load the kernel.

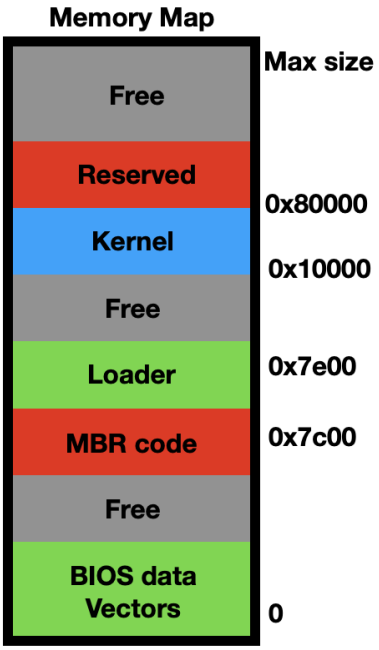

Figure 5: Kernel in memory

To play safe, the kernel is loaded at address 0x10000 because the loader is going to be extended later in the project and there will be enough space left to write a large kernel (until address 0x80000). The kernel is going to be quite large so there is going to be 100 sectors of data loaded which is roughly 50 kB. This is going to be enough for the kernel. Because the boot file covers the first sector and the loader the next 5 sectors, the kernel is written from the 7th sector. Before a new file called "kernel.asm" is created and loaded onto the image, the operating system needs to load and test some things before jumping to the kernel.

9.1 Get the memory map

One of the first parts that is needed is the memory map. The operating system wants the memory map because even though the memory segments are listed one by one in each part of this paper, the memory map can still vary among different computers so the free memory available for the system can be different. The BIOS interrupt call used to get the memory map is interrupt 15h. But before we call the services, we test if the services are available. To test the service, the register EAX is set to 0xe820h, edx is set to the ASCII code of smap (0x534d4150), ecx is set to 20 which is the length of memory block and in edi the memory address is saved in which we save the memory block. ebx should be 0 before we call the interrupt function (so we xor it with itself) because it is the first call. Now the interrupt call 15h can be called. If the carry flag is set for the first call to the function, the service e820 is not available (that's why we use CF which means "jump if carry

flag is set"). If it returns the memory info successfully, we continue to retrieve the memory info.

The BIOS interrupt call used to get the memory map is interrupt 15h which returns a list of memory blocks (20 bytes in size). The structure of these blocks is as follows:

Offset	Field
0	base address
8	length
16	type

The first 8 bytes (called a quad word or qword in the x86 language) is a 64-bit physical start address of the memory region. The second qword is the length in bytes of the region and the last 4 bytes (called a double word or dword in the x86 language) indicates the type of memory with 1 being free memory which we can use, 2 being not available to us, etc.... We only collect memory region of type 1. First off, the edi is adjusted, by adding 20 bytes to edi, to point to the next memory address (which is 20 bytes further because a memory block is 20 bytes). Then we pass the same parameters to EAX, ecx and edx. The ebx value isn't changed because ebx must be preserved for the next call of the function. Then the interrupt function is called and the carry flag is tested. Now if the carry flag is set this time, it means that we have reached the end of the memory blocks. If the carry flag is not set, edx is tested. If edx is nonzero, we jump back to get memory info. Otherwise we reach the end of the memory block and everything is done. After this the free memory information is known, the memory module will use the information to allocate memory for the system. Because there is no print function implemented (yet), the information of the memory map can't be printed yet.

9.2 Testing the A20 line

Next is to test the A20 line. The old machines have 20 bit addresses which memory can be addressed up to 2 of the power of 20 which is 1 MB. Later machines come with address bus wider than 20 bits. For compatibility purposes, the a20 line of the address bus is off so when the address is higher than 1 MB, the address will get truncated as if it starts from 0 again. This system runs on a 64-bit processor which has address wider than 20 bits. If we try to access memory with a20 line disabled, we will end up only accessing even megabytes, because a20 line is 0 and any address we are going to access is actually the address with bit 20 cleared. Therefore we need to toggle a20 line to access all the memory. There are multiple ways to activate the a20 line and each method is not guaranteed to be working across all the different machines. Newer machines have the a20 line enabled by default. To make this project simple, we assume our machines enabled the a20 line. The only thing needed to do is to test if the a20 line is enabled. The logic to test this is simple, if a20 line is not enabled and the system tries to access something where bit 20 is set to one in binary (for example 0x107c00 = 1 0000 0111 1100 0000 0000), then bit 20 will be 0. So the address that was actually referenced is the same address but with the 20th bit set on 0 (for example 0x007c00 = 0 0000 0111 1100 0000 0000). The example was chosen because this address was the start of the boot code and we can reuse this again for this test.

9.3 Setting up video mode

The last part to do before switching to long mode is to set up video mode. Printing on the screen is done by the BIOS service in real mode. Once the system is switched to long mode, the system is unable to call the BIOS functions. So a video mode is set so that characters can be printed on that mode. There are many video modes that we can use but in text mode is chosen because printing characters is simple in this mode. In another video mode (other than text mode), the layout and form of characters aren't defined and the developer has to make a character map for each character. A character is printed in text mode by specifying the ASCII code, the background and foreground on that character. Then the character will look like this on the screen:

Figure 6: Characters in VGA text mode

Text mode is set up by using BIOS interrupt 0x10, setting the parameter AH to 0h and then setting up the video mode, in this case text mode, by setting the AL register to 3h. The base address for text mode is 0xb8000 and the size of the screen that can be printed on is 80 in width and 25 in height (so 25 lines where there can be 80 characters printed on). The first position on the screen (first row and first column) corresponds to the first 2 bytes at address 0xb8000, the second position on the screen corresponds to the two bytes at b80002, and so on.... One character is thus 2 bytes long with the first byte the ASCII code of the character and the second byte are the attributes of that character. The lower half of the second byte is the foreground color and the higher half is the background color of that character.

Attribute								Character							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Blink	Background color			Foreground color				Code point							

Figure 7: Text buffer

A special attribute of a character is blink which can be enabled by setting bit 7 of the second byte to one. The character will blink when it's enabled. Depending on the usage, the blink can be ignored and the bit can be used as an extra bit for the background color which is done in the system. Because both background- and foreground color need to be specified in 4 bits, there are $2^4 = 16$ colors available which are listed below.

Index	Color	Index	Color
0	Black	8	Dark Gray
1	Blue	9	Light Blue
2	Green	A	Light Green
3	Cyan	B	Light Cyan
4	Red	C	Light Red
5	Magenta	D	Light Magenta
6	Brown	E	Yellow
7	Light Gray	F	White

10 Switching to protected mode

Now we jump to protected mode to prepare for long mode. To enter protected mode, the system needs to do a few things before entering protected mode such as loading the global descriptor table (GDT), loading the interrupt descriptor table (IDT), enabling protected mode by enabling specific registers and jumping to protected mode.

10.1 Global Descriptor Table

The global descriptor table (GDT) is a structure in memory created in the system and is used by the CPU to protect the memory. The GDT is a large table consisting of multiple blocks of memory called table entries. Each table entry describes a block of memory or segment and takes up 8 bytes.

The first entry should be empty or NULL. There can be more than over 8000 entries in the GDT but this operating system only needs at most 5 entries.

Figure 8: Global Descriptor Table

The GDT is pointed to by the value in the GDTR register. This is loaded using the LGDT assembly instruction, whose argument is a pointer to a GDT Descriptor structure.

79 (64-bit mode) 48 (32-bit mode) 16	15 0
Offset 63 (64-bit mode) 31 (32-bit mode) 0	Size 15 0

Figure 9: Global Descriptor Table Register

The size parameter is the size of the table in bytes subtracted by 1. This subtraction occurs because the maximum value of Size is 65535, while the GDT can be up to 65536 bytes in length (8192 entries). Further, no GDT can have a size of 0 bytes. The offset is the linear address of the GDT (not the physical address, paging applies). Note that the amount of data loaded by LGDT differs in 32-bit and 64-bit modes, the offset is 4 bytes long in 32-bit mode and 8 bytes long in 64-bit mode.

When the processor accesses memory, the value in segment register is actually an index within the GDT. This is different than in real mode where the value in segment register gets shifted 4 bits and added to the offset. In protected mode, segment registers (cs, ds, es and ss) hold the index instead of the base address. An example of an index is the following, the instruction mov EAX, [ds:0x1000] will copy the value in ds:1000 to register EAX. Suppose ds holds the value 16. Because each table entry takes up 8 bytes, the value 16 points to the third entry. The descriptor entry includes the base address of the segment, segment limit and segment attributes. Suppose the third entry has base address 0, then the base of this segment we are about to access is 0. Then we add the offset (0x1000) to the base which produces the address 0x1000. Because the descriptor entry includes the segment attributes it will tell the system if it has the rights to access the memory or it doesn't. If it doesn't, this operation will fail and the exception is generated. If all checks are done, the system can access the data in that memory location.

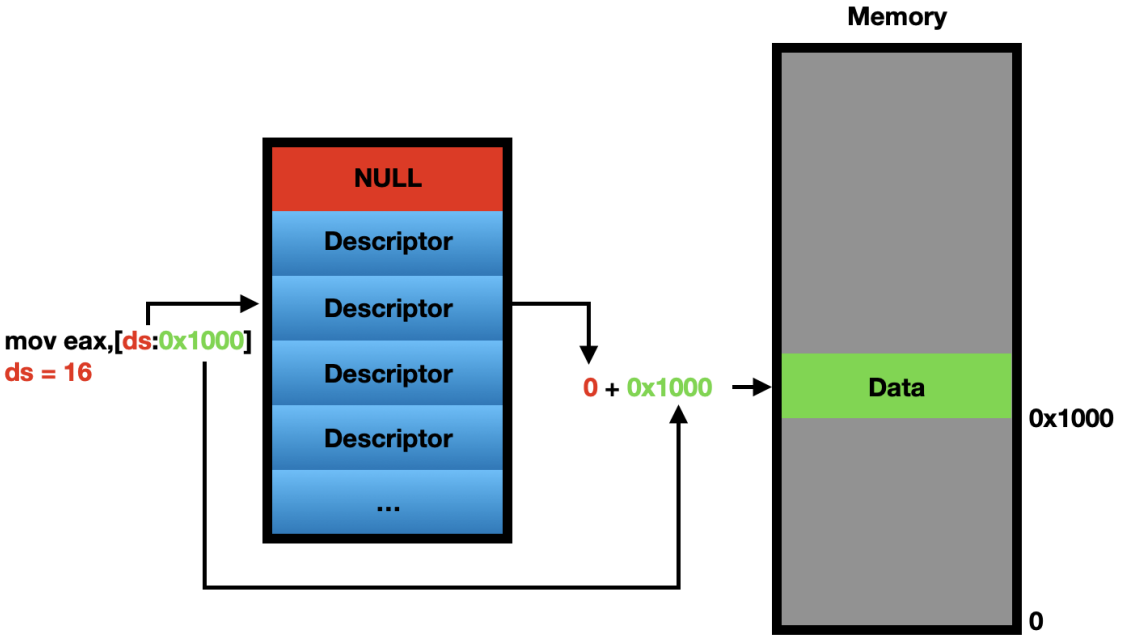

Figure 10: Descriptor addresses in memory

The structure of segment descriptor is as follows:

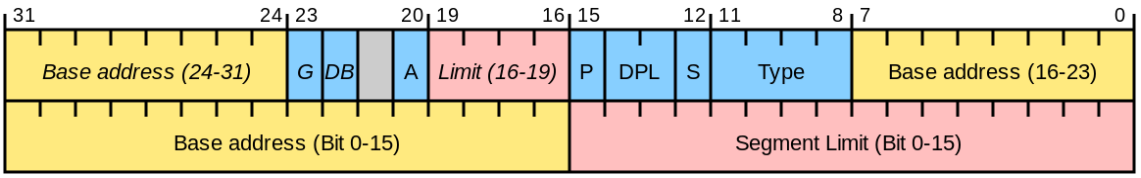

Figure 11: Segment Descriptor

- Base Address: 32 bit starting memory address of the segment.
- Segment Limit: 20 bit length of the segment. (More specifically, the address of the last accessible data, so the length is one more that the value stored here.) How exactly this should be interpreted depends on other bits of the segment descriptor.
- Granularity (G): If clear, the limit is in units of bytes, with a maximum of 220 bytes. If set, the limit is in units of 4096-byte pages, for a maximum of 232 bytes.
- Default operand size (D): If clear, this is a 16-bit code segment; if set, this is a 32-bit segment.
- Big (B): If set, the maximum offset size for a data segment is increased to 32-bit 0xffffffff. Otherwise it's the 16-bit max 0x0000ffff. Essentially the same meaning as "D".
- Long (L): If set, this is a 64-bit segment (and D must be zero), and code in this segment uses the 64-bit instruction encoding. "L" cannot be set at the same time as "D" aka "B".
- Available (AVL): For software use, not used by hardware
- Present (P): If clear, a "segment not present" exception is generated on any reference to this segment
- Descriptor privilege level (DPL): Privilege level (ring) required to access this descriptor

- Type: If set, this is a code segment descriptor. If clear, this is a data/stack segment descriptor, which has "D" replaced by "B", "C" replaced by "E"and "R" replaced by "W". This is in fact a special case of the 2-bit type field, where the preceding bit 12 cleared as "0" refers to more internal system descriptors, for LDT, LSS, and gates.
- Conforming (C): Code in this segment may be called from less-privileged levels.
- Expand-Down (E): If clear, the segment expands from base address up to base+limit. If set, it expands from maximum offset down to limit, a behavior usually used for stacks.
- Readable (R): If clear, the segment may be executed but not read from.
- A=Accessed: This bit is set to 1 by hardware when the segment is accessed, and cleared by software.

So when running in protected mode, setting up different segments with different privilege levels, the system can protect the essential data from accessing by user applications.

10.2 Interrupt Descriptor Table

Next off is the interrupt descriptor table (IDT) but first an interrupt is explained. An interrupt is a signal sent from a device to the CPU. If the CPU accept the interrupt, it will stop the current task and process the interrupt event. Different numbers are assigned to different interrupts. Also a handler or interrupt service routine is used so that the CPU will call that handler to do the specific task when an interrupt is fired. In order to find that handler, we use the IDT. The IDT is a data structure used to implement an interrupt vector table. The IDT is used by the processor to determine the correct response to interrupts and exceptions. The IDT could have a total of 256 entries and are also called gates, it can contain Interrupt Gates, Task Gates and Trap Gates. It is also a structure in memory but the fields of IDT entries are different from those of GDT entries.

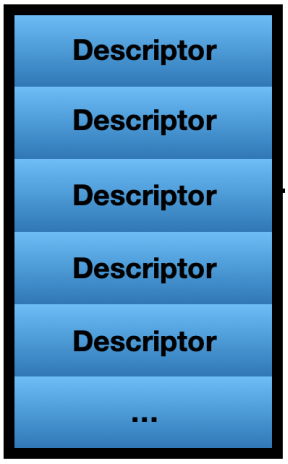

Figure 12: Interrupt Descriptor Table

The location of the IDT is kept in the IDTR (IDT register). This is loaded using the LIDT assembly instruction, whose argument is an IDTR. The parameter size is one less than the size of the IDT in bytes. The Offset parameter is the linear address of the Interrupt Descriptor Table (not the physical address, paging applies). Note that the amount of data loaded by LIDT differs in 32-bit and 64-bit modes, Offset is 4 bytes long in 32-bit mode and 8 bytes long in 64-bit mode.

79 (64-bit Mode) 48 (32-bit Mode) 16	15 0
Offset 63 (64-bit Mode) 31 (32-bit Mode) 0	Size 15 0

Figure 13: Interrupt Descriptor Table Register

Let's see an example of how the IDT works. Suppose the interrupt request number 3 is fired, then the cpu will get the corresponding item in the IDT (the 4th descriptor entry of the IDT since the table start with index 0). In the IDT entry is the information about which segment the interrupt service routine is at and the offset of the routine. IN this example, the base address of the segment this handler locates at is 0 and the offset is 0x5000. Then the system gets the address of the interrupt handler and the code of the handler is executed.

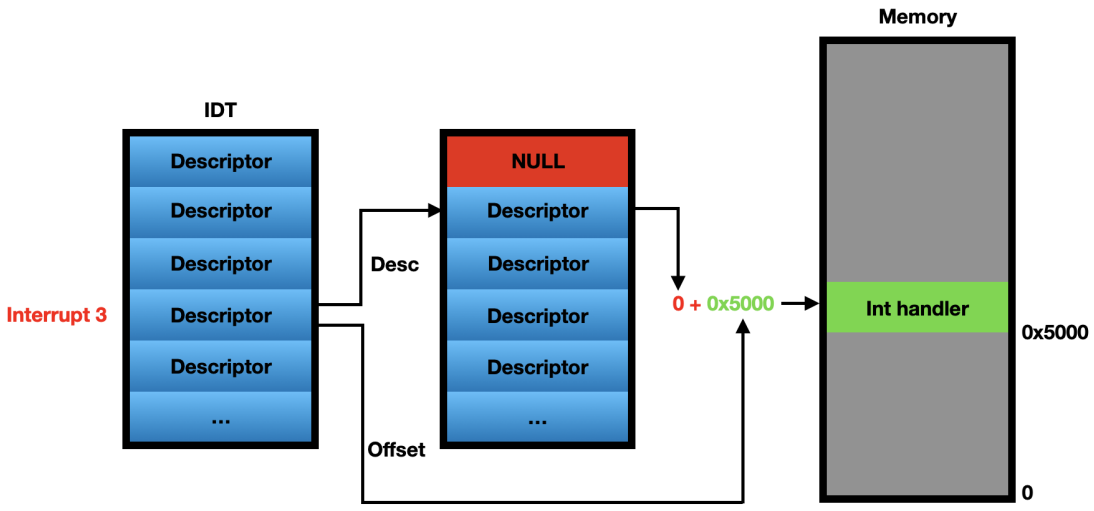

Figure 14: Interrupt addresses in memory

The structure of a gate descriptor is as follows:

- Offset: A 32-bit value, split in two parts. It represents the address of the entry point of the Interrupt Service Routine.
- Selector: A Segment Selector with multiple fields which must point to a valid code segment in your GDT.
- Gate Type: A 4-bit value which defines the type of gate this Interrupt Descriptor represents. There are five valid type values.
- DPL: A 2-bit value which defines the CPU Privilege Levels which are allowed to access this interrupt via the INT instruction. Hardware interrupts ignore this mechanism.
- P: Present bit. Must be set (1) for the descriptor to be valid

10.3 Privilege rings

In the x86 architecture, there are 4 privilege rings or levels from ring 0 to ring 3. Ring 0 is the most privileged level where all the instructions can be executed and all the hardware can be accessed.

Ring 3 is the least privileged level where only a subset of instructions can be executed and accessing hardware is generally not allowed. Although the CPU uses offers 4 levels of privileges, this system only uses 2 of them which is ring 0 in which the kernel runs and ring3 where user applications execute.

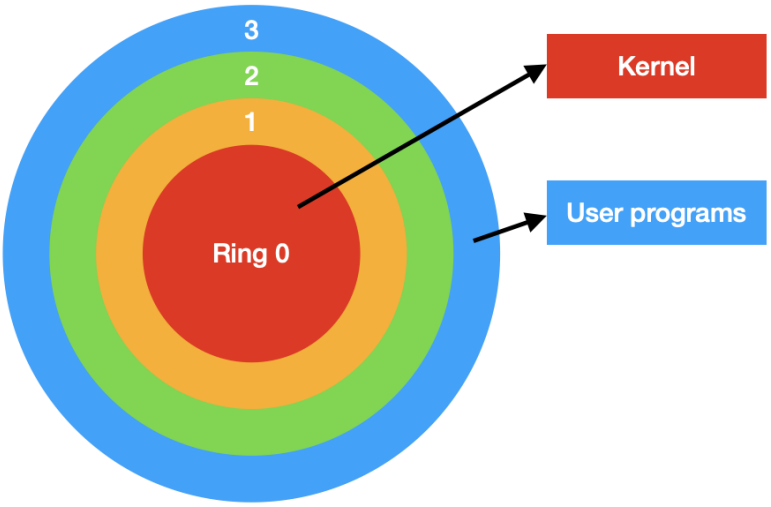

Figure 15: Privilege levels

The current privilege level (CPL) shows what level the system is currently running at and is stored in the lower 2 bits of cs and ss segment registers. When the lower 2 bits of cs and ss segment registers are 0, the system is in ring 0. If they are 3, the system is running in ring 3. In protected mode the CPL and requested privilege level (RPL) will compare against the descriptor privilege level (DPL). If the check fails, the CPU exception is generated. The DPL is stored in the descriptor table entries and the RPL is stored in the selectors.

10.4 Implementation

Everything will be implemented in the "loader.asm" file. The first thing that is done when a mode switch is happening is disabling the interrupt so that the processor will not respond to the interrupt. After switching to long mode, the interrupts will be re-enabled and processed. The instruction used for this is clear interrupt flag (cli).

Then we load the GDT and IDT structure. The register GDTR points to the location of the GDT in memory and we load the register with the address of the GDT and the size of the GDT. The instruction used for this is "lgdt" which takes one memory operand. This operand is the GDTR structure called "Gdt32Ptr" and their are two parts in this variable. The first two bytes are the size of the GDT minus 1 and the next four bytes are the address of the GDT as explained in the global descriptor table section. The GDT itself isn't defined yet so we define it. The start of the GDT is called "Gdt32" and where the first entry, which is zero, is defined. Remember that each entry is 8 bytes so we use the directive define quadword or "dq" which allocates 8 bytes of space. Next to define is the code segment descriptor which is defined in the label "Code32". The first two bytes are the lower 16 bits of the segment size. The code segment size is defined to the maximum size so it is set 0xffff. The next three bytes are the lower 24 bits of the base address which is set to 0 meaning that the code segment starts from 0. The 4th byte specifies the segment attributes.

Access Byte

7	6	5	4	3	2	1	0
P	DPL		S	Type			

Figure 16: System Segment Descriptor

The S shows if the segment descriptor is a segment descriptor or not. It's set to 1 meaning it is a code or data segment descriptor. The type field is assigned to the value 1010 in binary (or 10 in decimal) which means that the segment is a non-conforming readable code segment. The major difference between conforming and non-conforming code segments is that the CPL is not changed when the control is transferred to higher privilege conforming code segment from lower one. Conforming code segment is not used in our system. The DPL parameter indicates the privilege level of the segment. Because we want to be running at ring 0 when we jump to protected mode we set DPL to 0 and when we load the descriptor to the cs register, the CPL will be 0 indicating that we are at ring 0. The last parameter P is the present bit which means it has to be set to 1 when the descriptor is loaded, otherwise the CPU exception is generated. The total value is thus 10011010 which is 0x9a in hexadecimal so we assign this value to the 4th byte. The next bytes are a combination of segment size and attributes. The lower half is the upper 4 bits of segment size and it is set to the maximum size. The available bit (A) can be used by the system software which is ignored here (set to 0). The D bit which is the default operand size bit is set to 1. The granularity bit is set to 1 meaning that the size field is scaled by 4 kilobytes. Together this forms 1101111 in binary which is 0xcf in hexadecimal. The last byte is the upper 8 bits of base address which is simply set to 0. The data segment is the same only the type field is changed to 0010 in binary which means the segment is readable and writeable data segment. If we only set the segment to readable data segment, when we write data into the memory location referenced by this segment descriptor, the exception will be generated. The code en data segment descriptors are all we need in protected mode and we calculate the length of the descriptor table. We define a constant gdt32 length variable called "Gdt32Len".

Next is using the load IDT instruction. For this, we also need two structures, the IDTR and IDT. But since we don't want to deal with interrupts until we jump to long mode, we load the register invalid address and size zero. Note that there is one type of interrupt called non-maskable interrupt which is not disabled by the instruction cli. So when a non-maskable interrupt occurs, the processor willl find the IDT in memory. The CPU exception will be generated because the address and size of IDT are invalid in this case and eventually our system will be reset, which is what is wanted. The reason is that non-maskable interrupts indicate non-recoverable hardware errors such as RAM error, there is no need to boot our system if such error occurs. The only thing we do is reset the computer.

After we load GDT and IDT, we enable protected mode by setting the protected mode enable bit in cr0 register. cr0 is a control register which changes or controls the behaviour of the processor. Bit 0 is the protected mode enable bit. So the content of cr0 is copied to EAX and set bit 0 to 1 using an OR instruction. At last writing the value in EAX back to cr0. Now protected mode is enabled. The last thing that is done is loading the cs segment register with the new code segment descriptor we jsut defined in the GDT table. Loading the code segment descriptor to the cs register is different from another segment register. The mov instruction cannot be used to load the cs register so instead the jump instruction is used. The code segment descriptor here is the second entry which is 8 bytes away from the beginning of the GDT. So the selector that is used is with index 8, when the check passes the system is going to jump to protected mode and here the label for 32 bit is first defined.

In protected mode the other segment registers (such as AS, DS, ES and SS registers) are initialized. These are loaded with the data segment descriptors. The data segment descriptor is the third entry, so the index is 0x10. Then the stack pointer is set to 0x7c00.

Part V
Long Mode

Remember that there are two modes in 64 bit called 64-bit mode and compatibility mode but compatibility mode isn't used. So when there is talked about long mode, 64 bit mode is used. From now on when long mode is entered, the system will never leave it until the system is shut down. Preparing for long mode requires several steps including loading the GDT and IDT and enable paging. Note that paging is enabled before long mode is entered, since segmentation is disabled in 64-bit mode. Paging is used to perform memory protection. Then long mode is enabled by setting the specific registers and load the new code segment descriptor in the cs register using the jump instruction.

11 Before enabling long mode

11.1 Global Descriptor Table

The GDT is still used in 64-bit mode but the fields of the GDT entries are a bit different. Regarding segment registers, cs register is used, but the contents of ds, es and ss registers are all ignored. The memory segment referenced by those segment registers are treated as if the base address is 0 and the size of the segment is the maximum size the processor can address. generally, data segment descriptors are not needed but there is one special case where it is needed. That is, when we try to get from ring 0 to ring 3 to run user applications because getting from ring 0 to ring 3 requires to add a valid data segment descriptor to load into the ss register.

11.2 Interrupt Descriptor Table

The IDT is also still used in 64-bit mode but here each entry takes up 16 bytes, whereas in protected mode the IDT entry is 8 bytes long. The process of interrupt handling is the same as in protected mode. The structure of the segment selectors is not changed. If the systems wants to load a descriptor into a segment register, the index should point to a valid descriptor and CPL, RPL will compare against DPL in the descriptor. In 64 bit mode, it is rare to load segment descriptors by ourselves. In the system, the code segment descriptors are loaded when jumped to 64 bit mode and load the data segment descriptor in ss register when we get to ring 3 for the first time, which are the only two scenarios where the segment descriptors are loaded by ourselves.

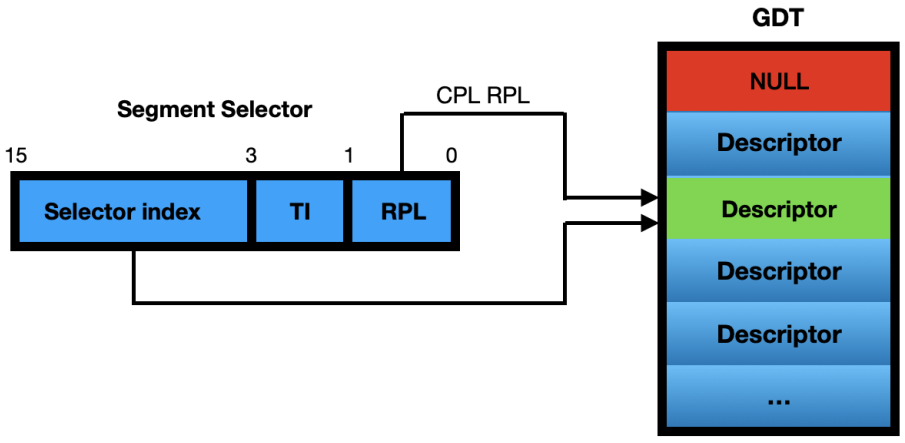

Figure 17: Segment Selector

Next, the descriptor entry format is explained in detail. The code segment will first be explained.

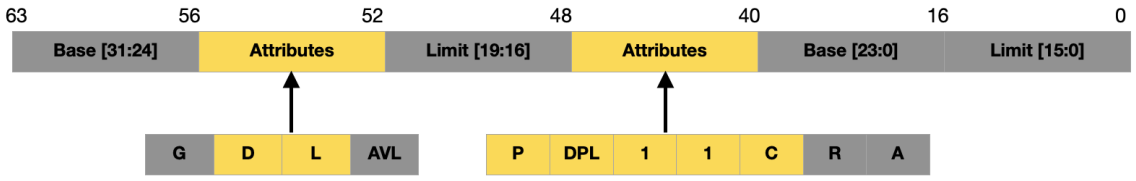

Figure 18: Code Segment Descriptor Entry

The fields that aren't highlighted are ignored which leaves only a few bits in attributes field to set. The fields which are set to 1 indicate that the descriptor is a code segment descriptor. The attribute "C" is the conforming bit, we only use non-conforming code segments in the system. The "DPL" attribute indicates the privilege level of the segment. It is set to 0, after the code segment is loaded into the cs register and jumped to 64 bit mode, the CPL will be 0 meaning that the system is in ring 0. The next bit is the present bit "P" which should be 1 otherwise the CPU exception is generated when the the descriptor is loaded. "L" stands for long which is a new attribute bit. If the bit is set to 1, it means that the system is running in 64 bit mode. If it is 0, the system would be running in compatibility mode. As for the "D" bit, the only valid value is 0 when the long bit is set to 1.

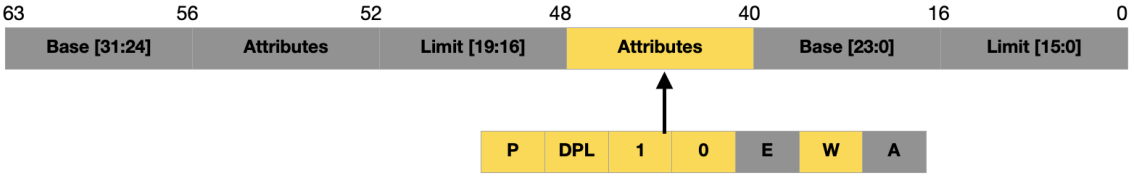

Figure 19: Data Segment Descriptor Entry

The data segment descriptor also have a few bits that can be set. The field "10" means that the descriptor is a data segment descriptor. "W" stands for writeable so it is set to 1. If the descriptor is loaded in the ss register with the "W" bit set to 0, then a CPU exception will be generated. "P" and "DPL" have the same meaning as those in the code segment descriptor.

Next is the memory address, where there are two terms called physical address and virtual address. The virtual address do not go directly to the memory bus. The virtual addresses need to be mapped to physical addresses by using the memory management unit (MMU). The physical address is the address put on the memory bus and then the data in that memory location is accessed. There are only a few cases where a physical address is used in the code directly. In 64 bit, there is 64 bits of virtual address space but not all the virtual addresses are available to the system. The canonical address is the address with the bits 63 through to the most-significant implemented bit set to either all ones or all zeros. For example, suppose a 48 bit address. Bit 47 is the most-significant implemented bit. If this bit is set to 1 the upper 16 bits, which is from bit 48 to bit 63, will all be set to 1. If it is set to 0, the upper 16 bits are set to 0. The value in the upper 16 bits will be considered as an invalid address or non-canonical address. When a non-canonical address is used, a CPU exception is generated. The code is based on the assumption that the processor only supports 48 bit virtual addresses so that the system can run on older machines.

Next off is setting up the memory map for the system in 64 bit mode. The user space and kernel space are the address space available to the system.

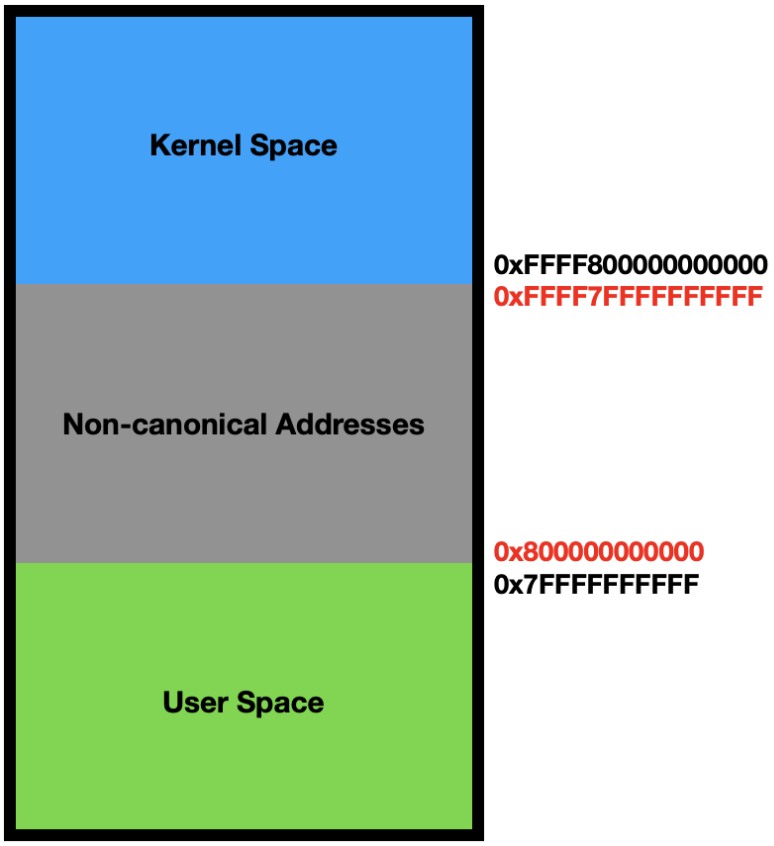

Figure 20: Virtual Memory Map

The other addresses are the address range where all the addresses are non-canonical addresses which cannot be used.

In the "loader.asm" file paging is the first thing that is set up. Enabling long mode requires us to setup and enable paging. The code is almost the same as the code for enabling paging in protected mode. This block of code finds a free memory area and initializes the paging structure. Then the GDT is loaded that will be used in 64 bit mode. Just like in protected mode, a GDT pointer and GDT is defined. The GDT in 64 bit is a bit different than the GDT in 32 bit. The first entry is empty so 0 is simply assigned to it. The second entry is used for the code segment. Setting it up is relatively simple because most of the fields are ignored.

D	L	P	DPL	1	1	C
0	1	1	0			0

Figure 21: Code Segment Attributes

The conforming "C" bit is set to 0 since non-conforming code segments are used in the system. The next 1's means that the descriptor is a code segment descriptor. The "DPL" attribute indicates the privilege level of the segment. The "DPL" is set to 0 when the descriptor is loaded to the cs register and jumped to 64 bit mode. The CPL will be 0 indicating that the system is at ring 0. The present bit "P" is set to 1, otherwise the CPU exception is generated if we try to load the descriptor. The long bit "L" should be 1 indicating that the code segment runs in 64 bit mode. The "D" bit should be 0 because the long bit is set. When all the fields are set, the value is assigned to the second entry.

The next thing is enabling 64 bit mode by setting the necessary bits in some registers. The first register is is register cr4. Bit 5 in cr4 is called the physical address extension or PAE bit. This bit has to be set to 1 before activating 64 bit mode. The next register is related to the cr3 register, the address of the page structure is copied, 0x80000 in this case, to the cr3 register. The address loaded to cr3 is the physical address.

Now long mode can be enabled. There is a model specific register called extended feature enable register and the eight bit is the long mode enable bit which should be set. The last thing is enable paging by setting the bit 31 in register cr0.

Part VI
Exceptions & Interrupt Handling

In this part it is explained how the system will handle interrupts and exceptions and finally how the system jumps to the kernel file. The kernel is also relocated from 0x10000 to the memory address 0x200000.

12 Relocating the kernel

The kernel is going to be relocated from 0x10000 to the memory address 0x200000. Remember that the system has already read the kernel file from disk into memory 0x10000. The direction flag is cleared with the instruction "cld" which enables the interrupts again (after the system disabled the interrupts with the "cli" instruction) and so that the move instruction will process the data from low memory address to high memory address (which means the data is copied in forward direction). The destination address of the kernel is stored in "rdi" register, in which the address 0x200000 will be stored and the source address is stored in the "rsi" register, in which the address 0x10000 will be stored. The kernel is 100 sectors long where each sector is 512 bytes long. Because the whole kernel needs to be moved, the instructions needs to loop $100 * 512$ times which can be done with the "rcx" instruction. But the instruction that is looped is a qword which is 8 bytes long. So the system has to loop only $100 * 512/8$ times. The "rcx" intsruction specifies that there has to be looped x times over every instruction that has the prefix "rep". After the instruction executes, the kernel is copied into the address 0x200000. With the last instruction (the "jmp" instruction), the system jumps to the memory address 0x200000. The execution is now transferred to the kernel.

To jump to the kernel, a "kernel.asm" file is made and add the corresponding commands (to convert to binary and add to the boot image). Here the kernel will manage its system resources in one place such as the GDT and IDT. So the GDT and IDT is loaded again in long mode (with minor changes because now everything is 64-bit and in protected mode is 32-bit).

13 Exceptions & Interrupts

When we talk about interrupts we actually refer to hardware interrupts such as timer, keyboard interrupts, ... Whereas exceptions occur as a result of exception errors or internal processor errors. The process of handling interrupts and exceptions are very similar.

13.1 Interrupts

How the interrupts are handled is already explained in part IV: Protected mode. There are 256 different exception and interrupt vectors that can be used.

Vector	Detail	Vector	Detail
0	Divide by 0	8	11 Segment
1	Debug	12	SS fault
2	NMI	13	GP fault
3	Breakpoint	14	Page fault
4	Overflow	15	Reserved
5	Bound	16	x87 FPU
6	Invalid opcode	17	Alignment
7	Device N/A	18	Machine check
8	Double faults	19	SIMD
9	Undefined	20-31	Reserved
10	Invalid TSS	32-255	User defined

The numbers are called interrupt vectors which identifies the exception and interrupts. Note that vectors 0 to 31 are predefined by the processor and can not be redefined. The vectors from 32 to 255 are user defined, the vectors for hardware and software interrupts are within this range.

In the "kernel.asm" file the IDT is defined which can have 256 entries total. The directive repeat is used ("%rep"), this takes one argument which is how many times we want to repeat the code that comes next (because we have 256 entries we want to repeat it 256 times). The code we want to repeat is between the "%rep" and "%endrep" statement and this are the entries in the IDT.
The attribute field is in the 6th byte and the value that is copied to this field is 0x8e which is 10001110 in binary (This field is the same as Figure 14 without the S bit). The present bit is set, DPL is set to 0 and type 01110 means that this is the interrupt gate descriptor. Every othe rfield is set to zero but will be changed later in the project. The IDT length is also added as a structure together with the IDT pointer. These structures are the same as with GDT. For each handler a label is written in the code. The handler ends with the "iretq" statement which is interrupt return. The interrupt return will pop more data than the regular return and can return to a different privilege level. It is worth mentioning that using the gate descriptor we can only jump from lower privilege level to a higher privilege level. The DPL of the code segment descriptor the selector 8 references is 0.

13.2 Saving registers

In the "kernel.asm" file the handler labels add push and pop instructions. The push instructions save the state of the CPU when an interrupt or exception occurs. When that happens, only some of the registers are saved such as rip and rsp registers. In the handler, registers are used and modified to perform specific tasks. Therefore when it is returned from the handler, the value of the registers may have changed of the CPU. It is not the same as it was before the entropic and the state of the CPU is not the same as it was before the interrupt occurs. So in order to restore the state of the CPU after the interrupt handling is done, all the general purpose registers are stored on the stack. Rsp is pushed on the stack when the CPU calls the handler, so here 15 registers are pushed.

After the interrupt is handled, the original value is popped in those registers and returned. A stack follows the LIFO principle (Last in, first out). So the value on the stack is popped in reverse order.

If the interrupt is fired, the handler will save all the general purpose registers first, do its work and then pop the value on the stack back to those registers. When there is returned from the handler, the previous wok the CPU was doing is resumed just like the interrupt never happened.

14 Programmable interrupt controller

The programmable interrupt controller (PIC) is used to manage the interrupt requests. So the PIC is first initialized before the hardware interrupt can be handled.

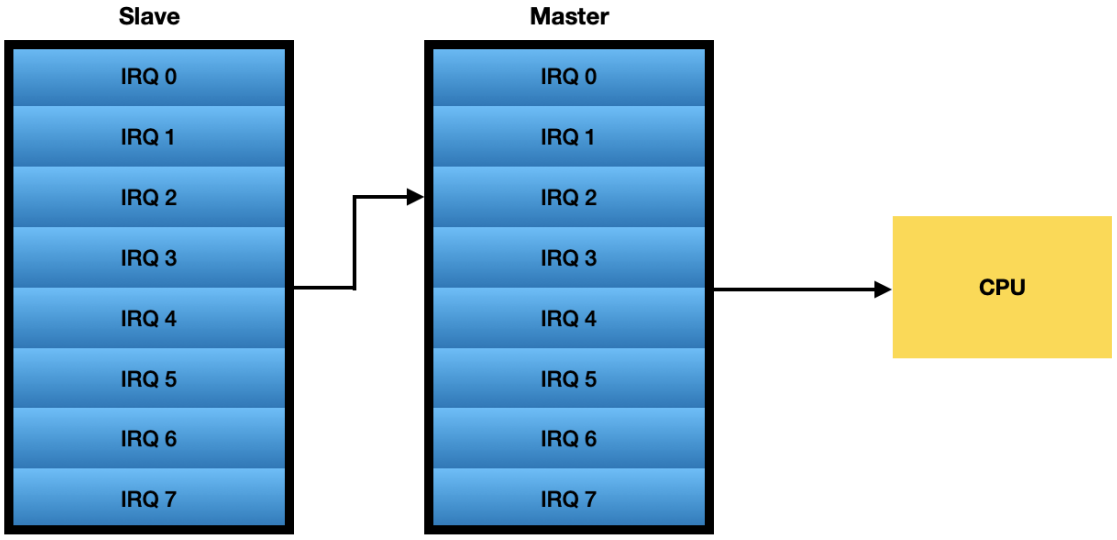

Figure 22: Slave & Master chip

There are two chips linked together. Each chip has 8 IRQ signals. The slave is linked to the master via IRQ2 so actually there is a total of 15 interrupts (for example, the programmable interval timer uses IRQ0 of the master chip and the keyboard uses IRQ1 of the master chip). When signal is sent to the interrupt controller, the controller will send signals to the processor according to the settings that are written in the controller. The processor will in it's turn find the handler for that interrupt by the vector number and call that handler.

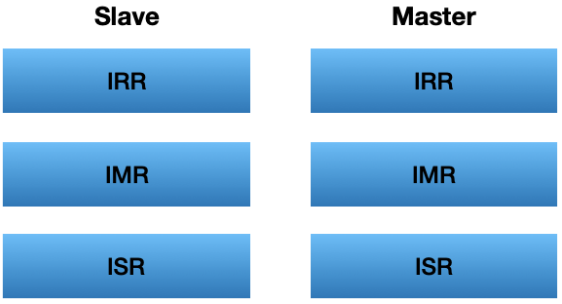

Figure 23: Chip Registers

There are three registers in each chip. These registers are used to service the interrupts and are 8 bit with each bit representing the corresponding IRQ. When a device sends a signal to the chip, the corresponding bit in interrupt request register (IRR) will be set. If the corresponding bit in the interrupt mask register (IMR) is 0, meaning that the IRQ is not masked, the PIC will send the interrupt to the processor and set the corresponding bit in the in-service register (ISR). Generally, there could come multiple IRQs at the same time which means more than one bit in the IRR will be set. The PIC will choose which one should be processed first according to the priority. Normally, IRQ0 has the highest priority and IRQ7 the lowest priority. Note that all the IRQs on the slave have a higher priority than IRQ3 or higher of the master (because the slave is connected via IRQ2 of the master).

14.1 Implementation

In the "kernel.asm" file, the PIT and PIC are initialized which require several steps. First the PIT.

There are three channels in the PIT, through channel 0 to channel 2. Channel 1 and 2 may not exist and these are not used in this system so channel 0 is the only channel that is talked about. The PIT has four registers, one mode command register and three data registers for the three channels respectively. The command and data registers are set to make the PIT work as expected, that is fire an interrupt periodically. The mode command register has four parts in it.

Bit 0 indicates that the value PIT uses is is in binary or BCD forms. It is set to 0 which means in binary form. Bit 1 through 3 is the operating mode (so there are 2^6 modes). It is set to mode 2 (010 in binary) which is the rate generator used for the reoccuring interrupt. Bit 4 to 5 is access mode. The data registers are 8 bits. If we want to write 16 bit values to the PIT, two bytes need to be written in a row. Access mode specifies in which order the data will be written to the data register, such as low byte only, high byte only, ... The access mode is set to 3 (11 in binary) which means the low byte is written first and then the high byte. The last part for selecting the channel with 0 being channel 0, 1 being channel 1 and so on. Since we only use channel 0, it is simply set to 0. Now we have the whole value of the register, which is 00110100, and it is moved to the al register. The PIT works like this, a counter value is loaded and the PIT will decrement this value at a rate of about 1.2 mega hertz which means it will decrement the value roughly 1.2 million times per second. In the system, the interrupt is fired at 100 Hertz which is 100 times per second. So the count value for the PIT is equal to 1193182/100 = 11931. Note that the counter is a 16 bit value. So first the ow byte of the value is written en the nthe high byte. The address of the data register of channel 0 is 0x40.

Next is the PIC. the PIC also has a command register and data register. Each chip has its own register set. The address for the command register of the master chip is 0x20 and the address of the slave is 0xa0. Bit 0 and bit 4 is set to 1. Bit 4 means that this is the initialization command followed by another three initialization command words. Bit 0 indicates that the last initialization command word is used. The first command word specifies the starting vector number of the first IRQ. The processor has defined the first 32 vectors for its own use. So the vector numbers 32 to 255 are defined here. The starting vector is thus 32. Instead of writing the data to the command register, it is written to the data register. The address of which is 0x21 for the master and 0xa1 for the slave IRQ. The next command word indicates that which IRQ is used for connecting the two PIC chips. On a regular system, the slave is attached to the master via IRQ2. If bit 2 of the word is set, it means that IRQ2 is used. So the value 4 is written to the data register. The last command word is used for selecting mode. Bit 0 should be set to 1 meaning that x86 system is used. Bit 1 is the automatic end of the interrupt but it is not used so it is set to 0. Bit 2 and 3 are also set to 0 which means buffered mode is not used. Bit 4 specifies fully nested mode but is also not used. This gives a total value of 1.

The interrupt controller works now but on thing has to be done. Since there are a total of 15 IRQs in the PIC, we only set up one device, the PIT. Therefore all the IRQs of the master and slave, except IRQ0 of the master, is masked. Masking an IRQ is done by setting the corresponding bit of the IRQ and write the value to the data register. So bits 1 through 7 are set to 1 and write is to the data register. Then the IDT entry is set for the timer which now can be used in the kernel.

15 Jumping from ring 0 to ring 3

The CPL is stored in the lower 2 bits of the cs and ss registers. The system is running in ring 0 so far which can be seen in the CPL. The code segment descriptor is prepared for ring 3 and the descriptor is laoded to the cs register. Once the descriptor is successfully loaded into the cs register, the system will run in ring 3. In this process, the data segment descriptor for the ss register is needed.

In the "kernel.asm", two more descriptors are added in the "Gdt64" structure. The first one is the code segment descriptor for ring 3. The value is pretty much the same as the second one, only 9 is changed to f which, as seen in binary, changes the DPL from 0 to 3. The other segment

will change this value also together with some attributes. The present bit is set to 1. The descriptor is also set to a data segment descriptor. The "w" bit is 1 indicating that the data segment is writable. Now, interrupt return is used to jump from ring 0 to ring 3.

To return to ring 3, 5 (8 bit) data is prepared on the stack.

Figure 24: Stack

The top of the stack is the RIP value which specifies where we will return, then the value of RFLAGS which contains the status of the CPU. When we return, the value will be loaded in RFLAGS register. The stack pointer is stored in the next location which will be loaded in register RSP. The last one is the stack segment selector. Because the stack follows the LIFO principle, the data is pushed in reverse order. The stack segment selector is on 0x18 so the descriptor that is referenced is the fourth one and the RPL is 3. Since we want to get to ring 3, the DPL of the descriptor and RPL of ss selector are set to 3. Now we are in ring 3.

16 Interrupts handling in ring 3

In this section we talk about the task state segment (TSS) and enable interrupts in user mode. When running in ring 3, an interrupt is fired. The control will be transferred to the interrupt handler which runs in ring 0 in the system. The DPL of the segment descriptor which the interrupt handler is at is set to 0. In this process, the ss segment register is also loaded with NULL descriptor by the processor. The rsp register is also loaded with a new value. The value of rsp is stored in the TSS. Therefore, the TSS is setup and the new value of rsp is specified in the TSS.

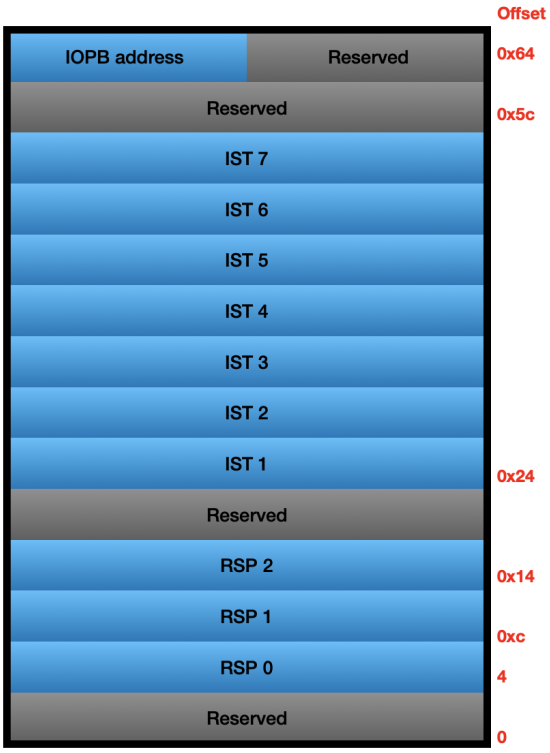

Figure 25: Task State Segment

This is the structure of TSS. The field of RSP0 is needed. When the control transferred from low privilege level to ring 0, the value of RSP0 is loaded into the RSP register. Since ring 1 and ring 2 is not used in the system, RSP 1 and RSP 2 fields are not set. The interrupt stack table is between the address 0x24 and 0x5c. If the IST field in the interrupt descriptor is non-zero, then it is the index of IST. For example, if it is 1, then the value of IST1 is loaded in RSP instead of RSP0 value. Since the interrupt descriptors are set with IST field being 0, the IST fields are not going to be used. The last item is the address of IO permission bitmap which is used for protection for IO port addresses. It is not used in this system.

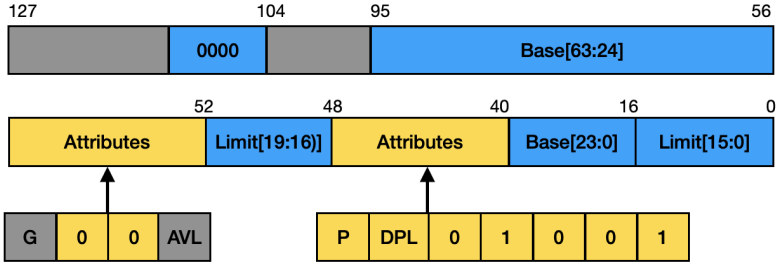

Figure 26: TSS Descriptor

The TSS descriptor is stored in the GDT. unlike the code segment and data segment descriptors where the base and limit are ignored, the base and limit attributes in TSS descriptor are used. So when the descriptor is set up, the base address and limit need to be set correctly. The value 01001 means that this is the 64 bit TSS descriptor. The meaning of P and DPL are the same as in the code and data segment descriptors. The G and available bit are not used in the system, so this part of the attribute will be set to 0. In the upper part of the descriptor, there are 5 zeros. The grey part is reserved. The TSS is already seen but we also need a selector to reference the descriptor. Loading the TSS selector is different from loading the code segment selector. Here the selector is

loaded using the instruction load task register. After the selector is loaded, the processor uses the selector to locate the TSS descriptor in the GDT.

16.1 Implementation

In the assembly file "kernel.asm", the TSS structure is defined. The first four bytes are reserved, so 0 is assigned to it. The next field is RSP0, this is the field that needs to be set with the new address of RSP which is 0x150000 when the interrupt handler is called. Other fields are not used in the system so they are simply set to 0. The last field is the address of IO permission bitmap. The size of TSS is assigned to it which means IO permission bitmap is not used.

Next is defining the TSS descriptor. In the GDT, a new entry is added called the TSS descriptor. The first two bytes are the lower 16 bits of TSS limit. The base address of TSS is the following two bytes. The address is defined in the C code so the next 3 bytes are set to zeroes as it are the lower 24 bits of the base address. The next byte is the attribute field. The value 0x89 is assigned to this field. The present bit is set to 1, DPL is 0 and 01001 specifies that this is the 64 bit TSS. As for the other fields, these are simply set to zero. The TSS is set and should work in the system.

16.2 Recurring interrupt

At this point, the interrupt is received only once. The timer interrupt is configured as a recurring interrupt which is fired every 10ms. If the handler returns instead of jumping to the end label, the interrupts will be received again and again. When handling a hardware interrupt, the interrupt needs to be acknowledged before we return from the handler, otherwise the interrupt will not be received again. To acknowledge the interrupt, the value to the command register of the PIC is written. It is written to the command register of the master. When the handler returns, the timer interrupt will fire periodically.

16.3 Spurious interrupt

A spurious interrupt is not a real interrupt. The IRQ used for spurious interrupt is the lowest priority IRQ number, that is IRQ7 for the master chip. The spurious interrupt is caused by several reasons and does not always occur.

In order to handle the spurious interrupt, the handler needs to be implemented for IRQ7. Because the slave chip is not used in the system, we only deal with the spurious interrupts in the master chip. The IDT entry is set first. Since there are three copies needed of the code to set IDT entries, the code is wrapped into a function "SetHandler" ("kernel.asm"). The address of the IDT entry is passed to the rdi register and the address of handler to RAX. Then the new function is called.

Another function called "SIRQ" is the actual handler. As with other interrupt handlers, the state of the CPU is saved in the beginning and restore its state before the handler returns. In the handler, the first thing that is done is checking if the interrupt is a spurious interrupt. To perform the check, the ISR register of the PIC is read and bit 7 is checked. If bit 7 is set, then it is a regular interrupt and it will be handled as other interrupts. Otherwise, this is a spurious interrupt and we simply return without sending the end of interrupt. To read the ISR register, a value is written to the command register.

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	1	0	1	1

Figure 27: ISR value

Bit 0 and 1 is to specify reading IRR or ISR. The value 11 in binary means reading ISR register. Bit 3 is set to 1 meaning that this is the command which reads the ISR register. Then the data is

read from the command register.

Part VII
Working with C

In this part the system is going to switch to C language to implement the kernel. But before the system can be switched to the C language, there have to be made some preparations first.

17 First C file

The first thing that was done in the "main.c" file that was created is adding the first function called "KMain" function where everything will be called and tested. The header files "stdint.h" and "stddef.h" are also inlcuded which would be explained further in this section.

17.1 Declaring data types

In the system we want to use fixed width data types. For simplicity, the system will assume the data has a specific length in the C code. For this, we use the "stdint.h" file which is a header file in the C standard library introduced in the C99 standard library to allow programmers to write more portable code by providing a set of typedefs that specify exact-width integer types, together with the defined minimum and maximum allowable values for each type, using macros. This header is particularly useful for embedded programming which often involves considerable manipulation of hardware specific I/O registers requiring integer data of fixed widths, specific locations and exact alignments. The types are in the form of intN_t for signed integers and uintN_t for unsigned integers. The exact-width integer types are as follows:

Specifier	Signing	Bytes	Minimum value	Maximum value
int8_t	Signed	1	-2^7	2^7
uint8_t	Unsigned	1	0	$2^8 - 1$
int16_t	Signed	2	-2^{15}	$2^{15} - 1$
uint16_t	Unsigned	2	0	$2^{16} - 1$
int32_t	Signed	4	-2^{31}	2^{31}
uint32_t	Unsigned	4	0	2^{32}
int64_t	Signed	8	-2^{63}	$2^{63} - 1$
uint64_t	Unsigned	8	0	$2^{64} - 1$

Note that there are also max and min types but these aren't used in the project.

The next header that is included in the "main.c" file is the header file called "stddef.h". The "stddef.h" header defines various variable types and macros such as size_t, which is the unsigned integral type and is the result of the sizeof keyword, and the macro NULL which is the value of a null pointer constant.

17.2 Calling Conventions

The first header that is created is a file to process interrupts. But before this can be done, some details about calling conventions are given. The system will use System v AMD64 calling conventions. The calling convention of the System V AMD64 ABI is followed on Solaris, Linux, FreeBSD, macOS, and is the de facto standard among Unix and Unix-like operating systems. The OpenVMS Calling Standard on x86-64 is based on the System V ABI with some extensions needed for backwards compatibility. In this calling convention, the first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, R9 (R10 is used as a static chain pointer in case of nested functions) and additional arguments are passed on the stack in reverse order. Integer return values up to 64 bits in size are stored in RAX while values up to 128 bit are stored in RAX and

RDX. RAX, RCX, RDX, RSI, R8, R9, R10 and R11 are called caller saved registers which means the caller has to save the value of these registers if it calls other functions which could alter these registers. Normally the caller will save the value on the stack before it calls other functions and restore the value of the registers after the functions return. The other registers RBX, RBP and R12 to R15 are called callee saved registers which means the value of these registers are preserved when we call a function and return from it.

The system needs the interrupt structures when it deals with the interrupts in C files. These structures are defined in the "trap.h" file. In the beginning of the header file we will add a guard so that the header file is included only once. If the label,

#ifndef _TRAP_H_
#define _TRAP_H_

is not defined then it isn't ensured that the contents of a given header file are not copied, more than once, into any single file to prevent duplicate definitions.

Because in this file structures will be defined using fixed width data types, the file includes the header file "stdint.h". Now the structures can be defined. The first one being the IDT entry. In the previous part (Exceptions & Interrupt Handling) the structure is defined as follows. The first two bytes stores the lower 16 bits of the offset, the selector is stored in the next two bytes, then a reserved field and type attributes which are both 1 byte long, the next two fields are the mid (16 bits) and upper (32 bits) bits of the offset and the last 32 bits are reserved.

The next structure is the IDT pointer which is used for the load IDT instruction. The first two bytes are the limit and the next 8 bytes are the address of the IDT. Note that the option attribute packed is added so that the structure is stored without padding in it. The packed variable attribute specifies that a variable or structure field has the smallest possible alignment. That is, one byte for a variable, and one bit for a field, unless you specify a larger value with the aligned attribute. If we don't add this attribute the data is stored to the natural alignment and we will have padding between these two items in this case. As a result, the load IDT instruction will load the register with wrong data. Before the last structure is defined, a new assembly file is created called "trap.asm". All interrupt related code will be added in this file. The handlers are declared globally so that they can be referenced in the C file. Each vector is an interrupt for something which is listed in the following table:

IVT offset	INT #	Description
0x0000	0x00	Divide by 0
0x0004	0x01	Reserved
0x0008	0x02	NMI Interrupt
0x000C	0x03	Breakpoint (INT3)
0x0010	0x04	Overflow (INTO)
0x0014	0x05	Bounds range exceeded (BOUND)
0x0018	0x06	Invalid opcode (UD2)
0x001C	0x07	Device not available (WAIT/FWAIT)
0x0020	0x08	Double fault
0x0024	0x09	Coprocessor segment overrun
0x0028	0x0A	Invalid TSS
0x002C	0x0B	Segment not present
0x0030	0x0C	Stack-segment fault
0x0034	0x0D	General protection fault
0x0038	0x0E	Page fault
0x003C	0x0F	Reserved
0x0040	0x10	x87 FPU error
0x0044	0x11	Alignment check
0x0048	0x12	Machine check
0x004C	0x13	SIMD Floating-Point Exception
0x00xx	0x14-0x1F	Reserved
0x0xxx	0x20-0xFF	User definable

The end of interrupt (eoi), read isr and load idt with all these other interrupts are processed in the same function called handler in the C file "trap.c". Since this function is not defined in the "trap.asm" file, extern "handler" is used in this file to call to the "handler" function.

!!! NOG EEN UITLEG GEVEN OVER ALLE VECTOR INTERRUPTS DETAILS IN TRAP.ASM !!!

The vector interrupts all call to the "Trap" function. This procedure is very simple. The state of the CPU is saved by pushing the general purpose registers. Then we call the "handler" function in the C file (in system v calling convention). The first argument is passed in register rdi and this passes the stack pointer to handler. When returned from this function, the end of "Trap" function is reached and the general purpose registers are restored. At last, the rsp register is adjusted to make it point to the correct location and the system returns. The rsp is added with 16 bytes because each vector operation pushes two values (both 8 bytes) on the stack before jumped to the "Trap" function to make the rsp point to the original location hen the exception or interrupt gets called by the processor. If the error code is pushed by the processor (like vector 8), there still needs to be added 8 bytes manually to skip the error code. There are also other procedures such as the end of interrupt, this sends the "eoi" command to the PIC, read ISR, which is required when processing spurious interrupt, and load IDT, because the IDT cannot be loaded with load IDT instruction directly in C. Instead we define a procedure "load_idt" and call this procedure in the C file to load IDT. This procedure has one parameter which is the address of IDT and is stored in register rdi.
The interrupt handler is now defined in the assembly file so it also needs to be declared in the header file "trap.h". This are all the functions of the form "void vectorX (void);" where X stands for an integer. They have no return value and parameters except for the last three (special) functions (init_idt, eoi, load_idt).
In the "trap.c" file the function which sets up the IDT is defined. There are two global variables defined called "IdtPtr" which is used when load IDT function is called and "IdtEntry", a vector array that forms the new IDT which includes 256 items/entries. The initialize IDT entry function takes three parameters, the address of the IDT entry, the address of the handler which is defined in the assembly file and the attribute of the IDT entry. In the function, we assign the value to

each field of the entry. The selector is hard-coded to 8 which is the kernel code segment selector. In the initialize IDT function, the IDT entry is initialized by passing the address of the IDT entry.

The only thing left to do is loading the IDT pointer. This is done by copying the limit of the IDT, which is the length of the IDT minus 1, and the address of the IDT. Then the IDT is loaded by calling the function load IDT. The last function is the handler function where all the interrupts are processed. The parameter is a structure called Trap Frame but actually it is the stack pointer.

When the handler is called, the state of the stack is like follows. The top of the stack is pointing to the value of r15. The general purpose registers, the index number and some of the error code are pushed manually in the assembly file. The rip, rsp, rflag value, ... are pushed by the processor. So the trap fram is defined according to it. At this point the data on the stack can be accessed by referencing the item in this structure.

In the handler, the index is checked which is pushed on the stack to see which interrupt or exception actually occurs. A switch statement is used for that. If the index is another exception or interrupt than the ones in the switch statement, the system will be stopped because there should be some errors in the kernel.

18 Simple library functions

A set of simple library functions can be created for the kernel. These will be added in the files "lib.c" and "lib.h". These functions are implemented using assembly language, so a file "lib.asm" is created.

18.1 Memory set

The first function is memory set which is pretty much the same as the one in the standard C library. Here, no value is returned. The first parameter is the address of the memory being set. The second parameter is the value the memory is set with. The size of the memory that is set, is specified in the last parameter.

In the assembly file, the instruction "cld" is used which clears the direction flag so that it copies data from low memory address to high memory address. Then ecx edx is moved and move al sil. The first parameter (of the C function) is passed in register rdi, the second one in rsi, third in rdx, fourth in rcx and so on. So rdi holds the address of the memory and the value that is copied is in register rsi. This value that is copied is 8 bits in size so it can be saved in "al" register. The third parameter which is the size is saved in rdx. The next instruction "rep stosb" will copy the value in al to the memory addressed by rdi register. The memory is now set so we can return.

18.2 Memory copy

This will copy a value in a memory address to another memory address. The first parameter of the function is the destination and the source is stored in the second parameter. The last parameter stores the size of bytes need copying. The memory move functionality is exactly the same so in the assembly file, memory move is a part of the memory copy code.

When data is copied from one place to another, there can be two situations. One is that the memory areas overlap and the other is that they do not overlap. If there is a scenario where the start of the source area is before the destination and the end is in the destination, the data has to be copied backwards. That is, from the last data block to the first data block otherwise the data at the back of the source will be replaced by the data in the front.

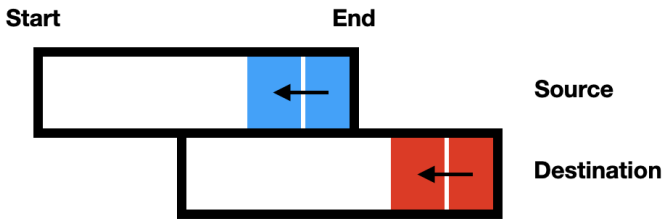

Figure 28: Overlap of addresses

As for the other scenarios, the data can simply be copied from the first to the last block. The cld instruction is also used and rsi is compared with rdi. Rdi holds the address of the source and rdi holds the address of the destination. The compare instruction used is "jae". If rsi is greater than or equal to rdi, we just jump to the copy part. If rsi is less than rdi, there is checked if the end of the source is in the middle of the destination. R8 rsi is moved and r8 is added to rdx which holds the value of the size. R8 is compared with rdi. If r8 is less than or equal to rdi, we jump to the copy part. If it is not, this is the situation the data needs to be copied backwards. A label "overlap" is defined to handle this special case. The first instruction is std or set direction flag which will copy the data from high memory address to low memory address. The the rdi and rsi registers are adjusted to the end of the destination and source so that the data can be copied from the end to the start. There is also decremented by 1 because otherwise it is one byte off from the correct position. Now the size can be saved in ecx. To finish, cld will clear the direction flag and return.

18.3 Memory move

This function is related to the memory copy. The first parameter of the function is the destination and the source is stored in the second parameter. The last parameter stores the size of bytes need copying. The rest of the assembly code is the same as the memory copy function.

18.4 Memory compare

This is the only one of the simple library functions that returns an integer. If the return value is 0, it means the result of comparison is equal. Otherwise they are not equal. The first two parameters are the addresses of the two memory areas being compared. The size is specified in the last parameter.

The cld instruction is also used and then the EAX register is cleared which will save the return value. The repeat while equal instruction will compare memory ans set eflags accordingly. If they are equal and ecx is non-zero, the process will be repeated. If the zero flag is set after execution this instruction, the result is equal otherwise it is not equal. The set instruction will set if the zero flag is cleared. If it is cleared, al is set to 1 which means the result is not equal. Otherwise EAX is set to zero meaning that the result is equal. Then the functions returns.

19 Print function

The print function is one of the most important functions in the system because this will allow for debugging when C code is written. It is not a library function so another file is created, called "print.h" and "print.c", to implement this functionality. This print function is a simplified version of the printf function used in the C language.

19.1 Implementation

In the "print.h" file, a macro is defined called "LINE_SIZE" which is set to 160. This is because a line has 80 characters in text mode and each character takes up 2 bytes. The structure screen

buffer is used for printing messages on the screen. The variable "buffer" is the buffer field which is the address of the screen buffer, 0xb8000. The next two items are the column and row which represents where the message will be printed next.

The print function in the system is called "printk" function which is used in the kernel. The first parameter is the address of the format string. It is const indicating that the data pointed to by format should not be modified. The function is a variable argument function which means it can take a variable number of arguments. There is a "..." at the end of the parameters which indicates that there are parameters but they don't need to be specified. Moving on to the "print.c" file, the printk function is implemented. A few variables are defined. The buffer is the array which can save a total of 1024 characters which will be printed on the screen. The "buffer_size" variable holds the count of the characters we want to print. The integer is used to save the value that needs to be printed on the screen. To make it simple, the integer that is printed in our system is assumed to be 64 bits. Next variable is a string pointing to characters. The last variable is called "args" where the type is "va_list" defined in the header file "stdarg.h". Now the args list is initialized using the "va_start" macro so that the variables can be used with other macros later. before the function returns, there is also done some cleanup using "va_end". In the rest of the function, the string is parsed. The format string is much like the string in the printf function in the C language. There are a few format specifiers that can be used in the printf function.

Specifier	Used for
%x	Hexadecimal integer (base 16)
%u	Unsigned Decimal integer (base 10)
%d	Decimal integer (base 10)
%s	String

The function will loop through each character in the format string. The format string here is null-terminated string. So if the null character is parsed, it means the end of the format string is reached. In the for loop, it is checked if the character is %. If the character is not %, it is a regular character and the character is simply copied to the buffer, the buffer size is incremented and moved to the next one. If it is %, the next character will be set. The switch statement will be used for this. If the next character is not defined in the switch statement, it is dealt with in the default. Since it is not a specifier that is supported, the character is simply copied to the buffer, the index is decremented and the loop is continued. If it is a specifier that is supported, the corresponding pointer is retrieved and characters are read from it. the macro "va_arg" is used to retrieve the pointer. Then a corresponding function is called to read the type into the buffer. The function will return the numbe rof characters it writes into the buffer. So the result is added to update the buffer size, then the for loop is breaked and the for loop is continued. After the for loop is exited, there are characters ready to write into the screen buffer. The function "write_screen" is called to do that.

In this function, the buffer size is needed which contains the characters, the buffer size, the address of the screen buffer and the character attribute. The structure includes the current position which tells us where to write the character. It is copied to the variable column and row. Then a for loop is used to copy the character to the screen buffer. The first if statement is used for scrolling the screen. If the character is not a newline, the character and attribute is just copied to the buffer. The address is stored in the field buffer of the structure. If it is a newline, the position will be changed to the next line so that the next character will be printed from there. The row is incremented and column is set to 0. Suppose there is no more room to print a character because the whole screen buffer is full, what is done is scrolling the screen.

If the row is greater than or equal to 25, the characters are copied which is in the second line through the last line to the first 24 lines and leave the last line empty. Memory copy is used to copy the characters. Then, memory set is used to empty the last line of the screen buffer. The row number is decremented since one line is moved up.

20 Assertion

The last function that is written in this part is the assert function which is very important in the system. The assertion is used in all the modules of the kernel. When different modules are used in the kernel, it is assumed that everything in it should run as expected. If there is a small error, this will bring the whole system down. The following picture is the structure of the system:

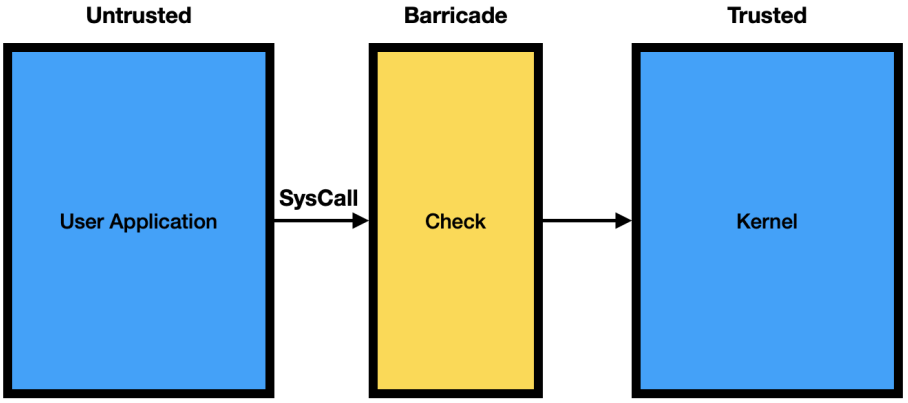

Figure 29: Assertion

The user programs are in the untrusted zone. When they send requests to the kernel by a system call, the barricade will check those requests and data to see if they are valid. If it is not valid, an error message will be returned to the programs. If the check passes, the kernel will process these requests and now it is in the trusted area where not the same checks are performed as in the barricade. Instead assertions will be used to find errors within the kernel. If the assumption made in the assertion fails, the assertion will stop the system and print the file name and line number which causes the error. So it could help finding some errors as the project is build up.

20.1 Implementation

The assertion will be implemented in the "debug.h" file and is a macro. The assert is defined and the parameter is the expression that is evaluated. If it evaluates to false, then the error check function will be called. The while loop is used to wrap up the macro, otherwise it may cause problems when used in the if statements. The value in the while loop is 0 so that an if statement only executes once

The error check function "error_check" takes two parameters. The predefined macro file and line to it are passed. The file will expand to the current file path and line expands to the current line number. in the C file "debug.c", the function is implemented. There are only two things done, printing the message and jumping to the infinite loop. The message includes the name of the file and the line number. After this is printed the code will jump to the while loop to stop the system. The assert can be used in the code now.

Part VIII
Memory management

In this part the memory manager is build and paging is set up. The free memory is normally above the address 0x100000 and the kernel is running at starting address 0x200000. That means that everything between 0x100000 and 0x200000 is available as free memory. But not all this free space is available to use. To know this, the memory map is found. To find the information about the memory map, code is implemented in "loader.asm". The memory map is stored in address 0x9000.

21 Retrieving the memory map

21.1 E820

The data that is retrieved is stored in structures (found in the "E820" struct in the "memory.h" file). The first structure is 4-byte data and is initialized with value 0. This is the counter of the structures. The real structures are stored from address 0x9008. In the code, we keep looping over structures until we reach the last structure. This is done by testing ebx. if ebx is 0 it means that the end of the structures is reached and the system will jump to the label GetMemDone, if ebx is not 0 it means the last structure is not yet reached and a new memory block is retrieved. Meanwhile the counter of structures will be updated each time. The structures are defined in detail in the "memory.h" file.

The first structure that is defined is a structure that retrieves memory info using BIOS service and is called E820. The first field in the structure is the base address of the memory region. The second field is the length of the memory region. And the last one is the memory type. The free memory region that can be used is the memory region with type 1. Notice that we add attribute packed so the structure is stored without padding it. Otherwise the structure will be larger than 20 bytes and the data will not be interpreted correctly in memory.

The next structure is called "FreeMemRegion" or free memory region because some of the memory regions are not available to use. This structure is thus used to store the information about the free memory. This structure only has two fields in it. The base address and the length of the memory region.

21.2 Printing free memory regions

Next off is looking to the "memory.c" file. The free memory region struct is used to store information for later use. The system assumes that there would be no more than 50 blocks of memory regions. The function "init_memory" will hold some variables. The first variable holds the number of memory regions which is stored at address 0x9000. The next variable "mem_map" holds the address of the data retrieved by the BIOS services. The base address is at 0x9008. Each memory region is stored in a E820 structure. The variable "free_region_count" is used to store the actual number of free memory regions. The other variable "total_mem" holds the size of the total free memory that can be used in the system. The variable free region is actually an array of free memory region structures. If the type of a free memory region is 1, then it can be used in the system and it is printed in the kernel.

22 Paging

22.1 Remapping the kernel

The kernel space until now was running on address 0x20000. The kernel is remapped to the address 0xFFFF800000000000. The lower part from address 0x800000000000 to 0xFFFF7FFFFFFFFFFF, are non-canonical addresses. The user programs will be run in the lower part of the memory location (from 0 to 0x7FFFFFFFFFFF). So the non-canonical address area separates the kernel and the user space.

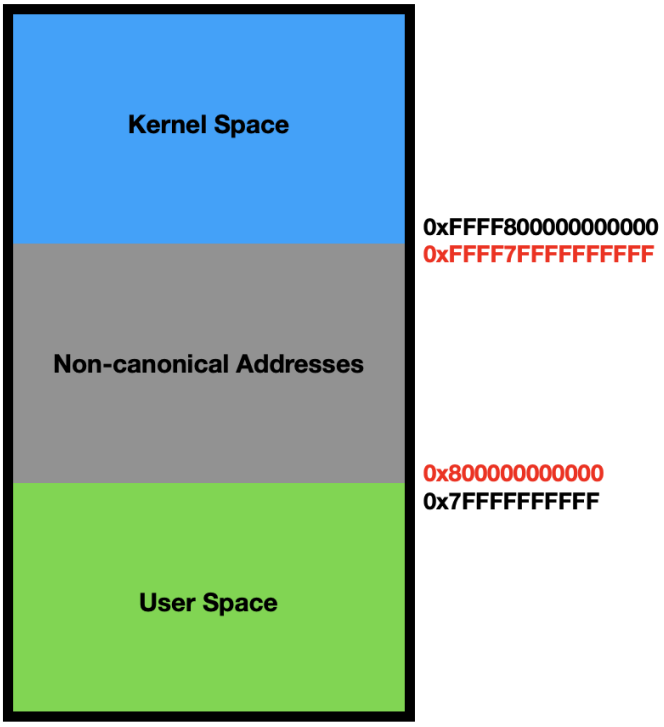

Figure 30: Kernel in virtual memory

To do all this, we need to set up paging. When 64-bit mode is entered, paging mechanism is enabled. But the way paging is set up is making the virtual addresses equal to the physical addresses. Therefore it needs to be reconfigured to do the job.

22.2 Pages

Paging allows the system to remap data into physical address space using fixed size blocks called physical pages. The page sizes the processor supports in 64-bit mode are 1GB page, 2MB pages and 4KB pages. In this system, we use 1GB and 2MB pages. So the memory address can be divided into different sets of pages. Page translation will use the data structure called page translation table to translate virtual pages into physical pages. A page can be translated into different pages or can be translated to the same page.

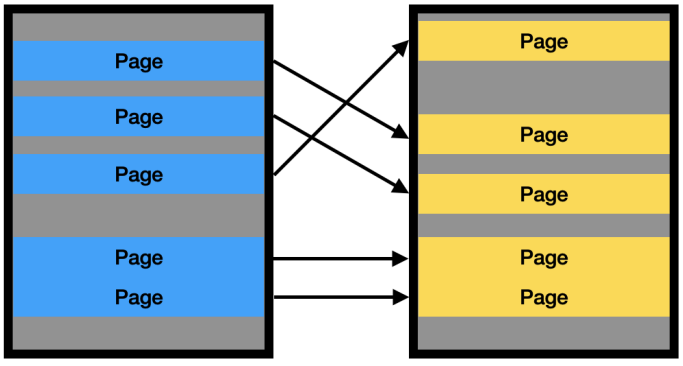

Figure 31: Page translation

22.3 Translation table

The translation table is actually a hierarchical data structure which means we will have several tables. The virtual address is broken up into three pieces.

First off is the 1GB page. The sign extension is not used. We start from page map level 4 table (PML4). The table is pointed to by the register cr3. Then the 9-bit PML4 value is used as an index to locate the corresponding item in the table. Each entry here includes the address of the next lower level table. Then the value of PDP is used as an index to locate the correct entry. The entry now points to the physical page which is 1G page. The page offset is used as the offset into the physical page and now the physical address is retrieved.

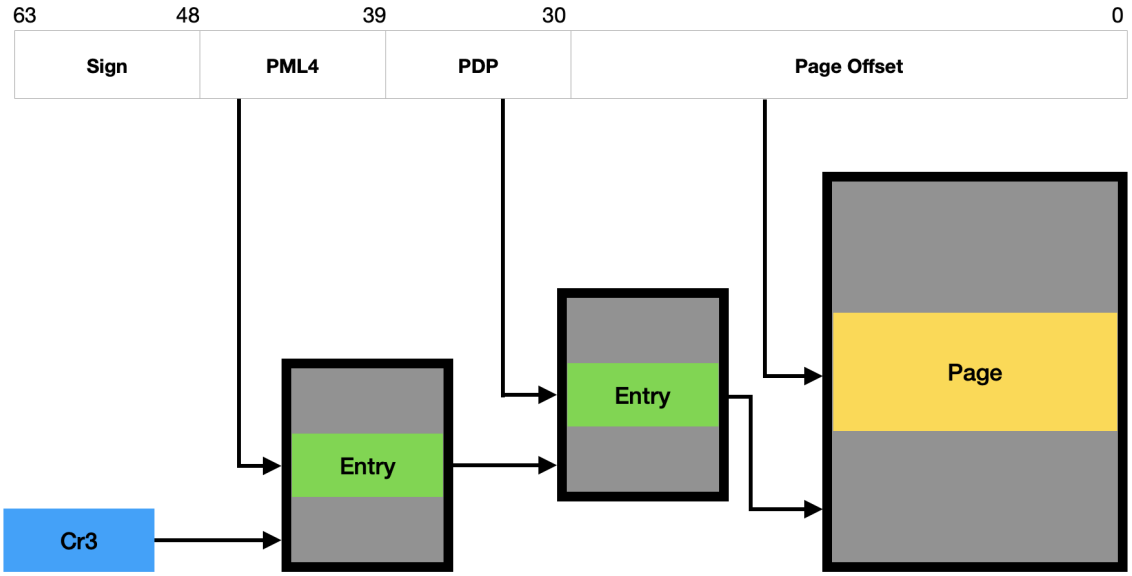

Figure 32: 1G Page translation

In 2 megabyte, the virtual address is divided into more fields but the rest of the process is the same. Register cr3 holds the address of PML4 table and PML4 acts as an index to the entry which points to the next level table. Then the value of PDP is used to locate the corresponding entry. At this point there is one more level. The next level table is called page directory table (PD) and the entry can be found using the PD value. These are the entry points to the 2MB physical page. The page offset is the offset into the page. Note that the sign extension is also not used and the addresses used in register cr3 and table entries are all physical addresses.

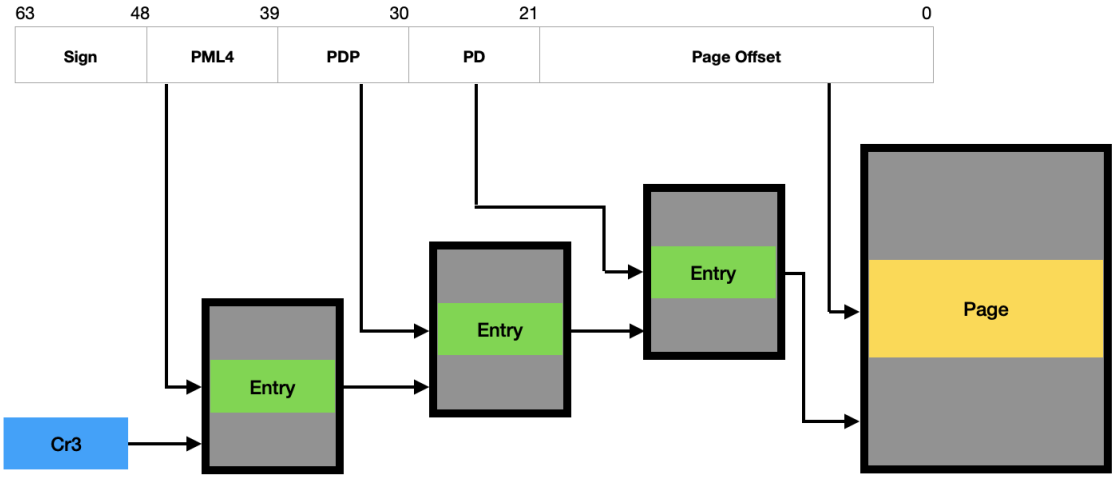

Figure 33: 2M Page translation

22.4 Table entries

There are three table entries that are used for 1GB page translation. There are other attributes that will not be discussed here because these are enough to let the memory module work (so only the attributes in the following picture).

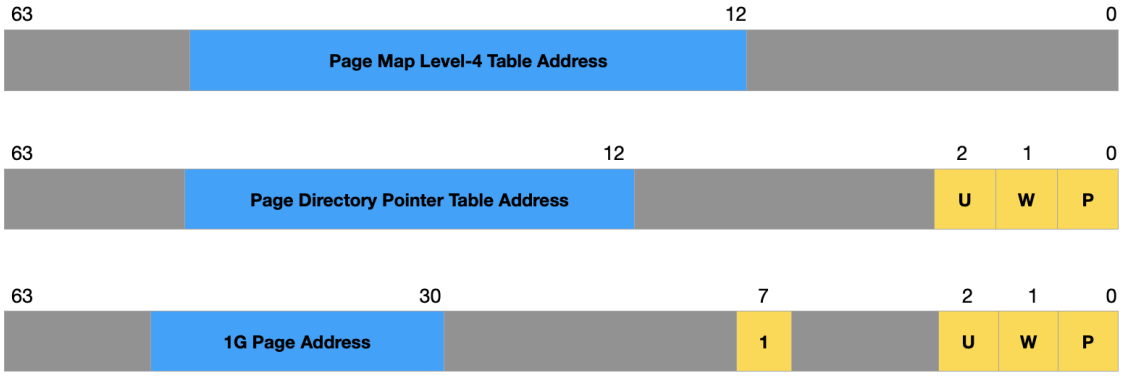

Figure 34: 1G Table Entries

The first entry is stored in register cr3 so the address of the page map level 4 table (PML-4) is saved to the register. The next entry is PML-4 table entry which holds the address of the page directory pointer table. In addition, there are some attributes in the entry such as bit U, W and P. The bit P represents present. If the bit is 0, it means the page table or page is not in the memory and a page fault exception will be generated if it is accessed. In our system it is set to 1. The W bit means read/write with 0 being read only and 1 being read and write. Bit U means user. If the bit is 0, the user program in ring 3 is not allowed to access the page. If the bit is 1 then the user program in ring 3 can access the page. The last entry, which is the page directory pointer table entry, holds the address of the 1GB physical page. This entry also has a few attributes, namely bit U, W and P. These attributes have the same meaning as the previous entry. Bit 7 must be 1 indicating a 1GB physical page translation.

Moving on to the 2MB page translation.

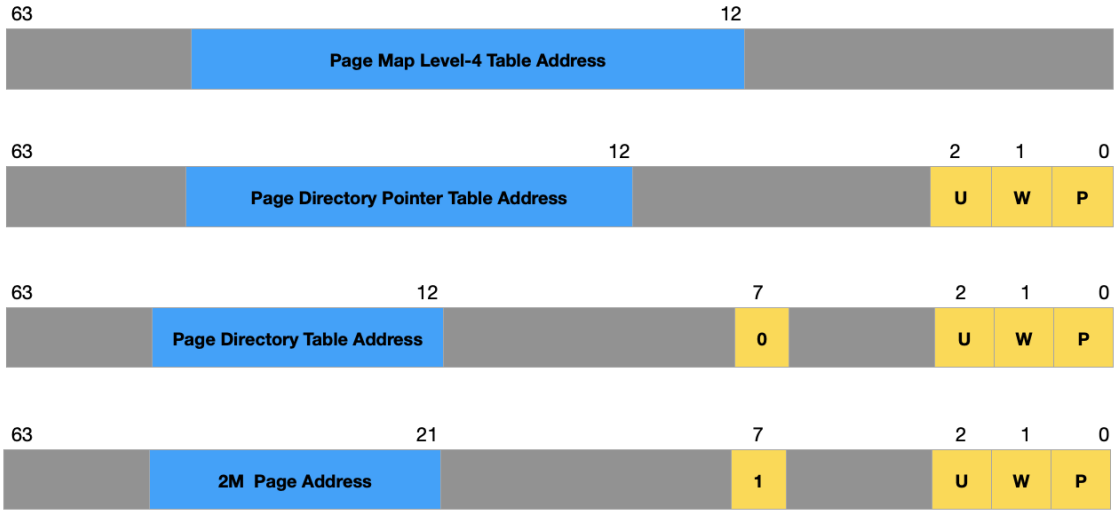

Figure 35: 2M Table Entries

The first two entries are the same as those in the 1GB page translation. The third entry is pointing the page directory table instead of 1GB physical page so bit 7 is now cleared. The last entry points to the 2MB physical page where bit 7 is set to 1 indicating this is a 2MB physical page translation. The last three entries have bits U, W and P as attributes which are exactly the same as the attributes from the 1GB page translation entries.

22.5 Implementation

To set up paging, code is added in the "loader.asm" file. First off, 10000 bytes of memory region is zeroed starting from address 0x70000. The value of cr3 is 0x70000 so the address of page map level 4 table is 0x70000. Each entry in the page map level 4 table represent 512 gigabyte but this system only implements the low 1 gigabyte. The first table is also implemented. Each table takes up 4 kB space, since the table includes 512 entries with each entry being 8 bytes. So the next table address is set to 0x71000 and the lower 3 bits are the attributes that need to be set. The memory needs to be readable and writable and only accessed by the kernel. Also, the present bit is set to 1. We get the following:

U	W	P
0	1	1

So the value is 0x3. When we add it to the entry we get 0x71000 + 0x3 = 0x71003. In the page directory pointer table, the first entry is set up. Because each entry here points to 1GB physical page, the entry at address 0x71000 is the entry that the system want to set. The base address of the base address of the physical page is set to 0 and the attribute here is set to the same value 0x3. Bit 7 is set to 1 to indicate that this is a 1GB physical page translation.

The next thing that is done is remapping the kernel. The virtual address that the kernel is currently running at is 0x200000 and the address the kernel is remapped to is the virtual address 0xFFFF800000200000. The virtual address is mapped to the same 1GB physical page. So the two translations that are set up will eventually point to the same physical page where the kernel is stored. Which means we just change the virtual address of the kernel by simply setting up the table instead of copying the kernel. So for the virtual address that is wanted, a simple calculation needs to be done. The virtual address is beyond 1GB so the 9-bit page map level 4 value is retrieved located at bit 39 of the address. Next, the other bits are cleared using the AND instruction and then the 9-bit index value is gotten. Now, using the index the system is going to locate the corresponding entry in the table. Each entry take up 8 bytes, so the index is multiplied by 8. The value we want to assign is 0x72003 which means the next level table is at 0x72000 and the attributes

are the same as before. The 1GB physical page is set to the same physical page where the kernel locates because we don't want to copy the kernel elsewhere. Then the kernel is relocated. Since the kernel is relocated to the new virtual address which is far away from the loader, it needs to be saved to the 64-bit register and jumped to the kernel. So the virtual address is moved to RAX and is then jumped to.

Jumping to the "kernel.asm" file, the first thing to do is change the load GDT instruction. Since the kernel is running at the high memory location, the memory address of the GDT pointer is also at this memory area. We cannot reference it using 32-bit address so instead the address is moved to register RAX and load RAX. Next is the kernel entry. Because the push instruction can only push 32-bit immediate values which cannot represent the address of the kernel entry. The address is moved to the 64-bit register and then the value is pushed on the stack. The address of the kernel entry is moved to RAX and then pushed. Next one is the stack address. The stack address is copied to the rsp register is at the low memory location. It is changed to the high memory address which points to the same physical address. The virtual address that is setup in the loader file points to the same physical page. So adding 0x200000 to the new virtual address is the new stack address we want.

23 Memory allocator

The free memory that is collected is divided into a bunch of 2MB pages and save them in the linked list for later use. The free memory here is physical memory. These pages are used for the memory module. So the memory module uses 2 MB pages instead of 1 GB pages.

23.1 virtual to physical address

The problem is that the kernel is now mapped to the higher part of the memory, and virtual addresses are used in the system after we enter 64-bit mode. The free memory the memory module uses is physical address. The first thing is to find the physical address according to the virtual address.

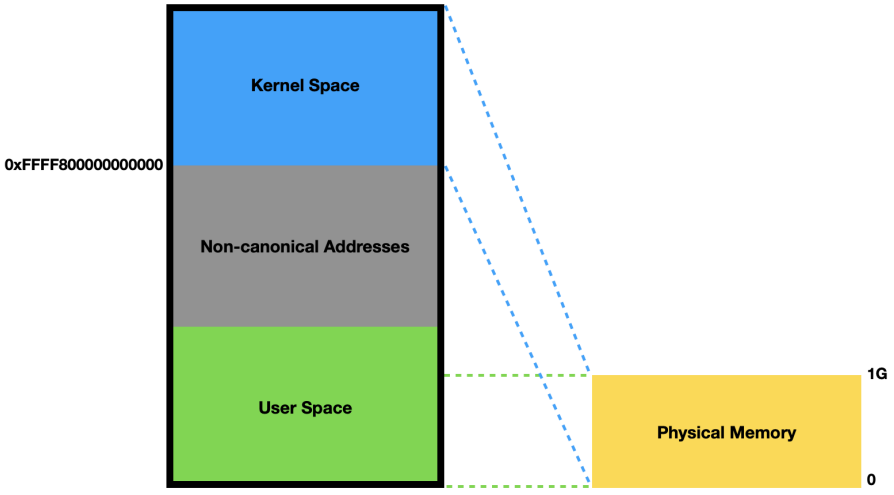

Figure 36: Mapping to physical addresses

In the picture we see that the lower 1GB address space is mapped to the corresponding physical page. We also map the kernel which is located at the higher part of the virtual space to the same physical page. The whole 1GB of memory space is mapped to the 1GB physical space, which means the offsets within the memory space remain unchanged. To get the physical address we can simply subtract the value which is the base of the kernel space from the virtual address, like the following equation:

virtual address = physical address + 0xFFFF800000000000

The first 1GB of RAM is used and the other free memory is simply ignored. The reason is that only 1GB of RAM is mapped in the loader file ("loader.asm") and 1GB of RAM is enough for the systems small kernel. In the "memory.h" header file, five macros are defined. The first one is the page size which is 2 MB. The next two macros align the address to 2MB boundary. The page align up will align the address to the next 2 MB boundary if it is not aligned. The page align down will align the address to the previous 2 MB boundary. The "P2V" and "V2P" macros converts between virtual address and physical address by adding or subtracting the base of the kernel.

In the "memory.c" file, the free memory is divided into a bunch of 2 MB pages. In this function two variables are defined "vstart" and "vend" which represent the beginning and the end of the memory region. The macros "P2V" and "V2P" are used to convert the addresses. These variables hold the virtual addresses instead of the physical addresses.

23.2 Initialization of free memory

In this system, the memory that is cared about is from the end of the kernel to the end of the first gigabytes of memory. To know the end of the kernel, a symbol is added in the linker script ("link.lds"). At the end of the linker script, PROVIDE(end = .) is written, which means the label "end" will represent the current position that is the end of the kernel. At this point the label "end" can be used in the C file. In the C file "memory.c" extern end is added since it is not defined in this module and set to char type.

The next thing is checking the location of the memory region. If the start of the memory region is larger that the end of the kernel, this is the free memory that is wanted and the initialization is done by calling the function "free_region". This function divides the region into 2 MB pages and the function "kfree" is called to collect the pages.

The "kfree" function will do the initialization. Before something is done here, simple checks are added. The first one is to make sure that the virtual address is page aligned. The next one assumes that the virtual address is not within the kernel. The last one checks 1G memory limits. The bunch of pages are stored in the form of a linked list. When a page is added, what is really done is link the page to the existing one. As more pages are added, the list will be extended. This is done in reverse order.

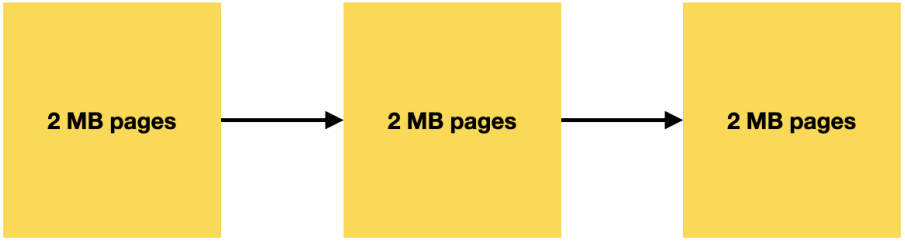

Figure 37: Linked list of pages

In the C file "memory.c" a variable "free_memory" is defined which is the head of the linked list. In the function, the virtual address is converted to the pointer to type structure page. Next, the head of the list is copied to the first 8 bytes of the page and then save the virtual address to free memory. At this point, the head of the linked list points to the current page.

Note that the address of "end" is needed ("&end") because "end" itself is a symbol instead of a variable. If the start address of the memory region is less than the end of the kernel, the end of

the region will be compared with the end of the kernel. If it is larger than the end of the kernel, the memory will be initialized from the end of the kernel to the end of the region. Otherwise the region is ignored because it is within the kernel. After the loop, the free memory pages is collected and a variable memory end to save the end of the free memory. Because pages are added to the list in reverse order, the head of the list points to the last page that is collected. Then the page size is added.

The function "kalloc" is used in kernel mode. Allocating memory is just removing a page from the page list and returned to the caller. The first page in the list is copied to the variable "page_address" and check to see if it is NULL. If the page is not NULL than this is a valid page and we will return the page to the caller. The head of the list will then point to the next page. Now the page is removed from the list.

24 Memory pages

Now that the free memory pages are collected and saved in the linked list, the kernel is again remapped to the same address space as before but now 2 MB pages will be used instead of 1 GB pages which was used by the kernel so far. To do the kernel remap, the corresponding table entries need to be set up. The tables that are now used are the PML4 table, page directory pointer table (PDPT) and page directory table (PDT). The entry in the PDT will point to the 2 MB physical page.

24.1 Implementation

The C file "memory.c" will start from the function "init_kvm" which will call the function "setup_kvm".

When this function is called, a new free memory page can be allocated using the function "kalloc". This page is actually the new page map level 4 table. If the allocation failed, the system is just stopped. If the check passes, the page is zeroed using the "memset" function because the page could include random values when they get the page from the function "kalloc". The function "map_pages" takes a few parameters. The first one is the PML4 table. The next two parameters are the start and the end (virtual) address of the memory region we want to map. The next one is the start of the physical page we want to map into. Because the kernel has to map to the same physical address, the physical address of the kernel base is passed. The last parameter holds the attribute of the page table or physical page. If the mapping fails, it returns false, else true.

The next function is the map pages function ("map_pages"). In this function, two variables are defined named "vstart" and "vend" which save the aligned virtual addresses. If an unaligned starting virtual address is asked, the page needs to be mapped where the start address is located. So the the address is aligned to the previous 2 MB page so that the start address can be included. The same thing happens with end address. Then the variable PD is defined which is used to set the page directory entry. The index is used to locate the specific entry in the table. After that, the mapping is done. the first thing that is done is finding the PDPT which points to the page directory table by calling the function "find_pdpt_entry". If it returns null, it means that it failed and false is simply returned. If it returns a valid table, the index is used to locate the correct page entry according to the virtual address. Then the entry is set with the physical address and attributes.

The function "find_pdpt_entry" that is used in the previous function takes a few parameters. The PML4 table, a virtual address and "alloc" indicating whether or not a page is created if it does exist. The last parameter are the attributes. In this function a few variables are defined. Since we find the PDPT entry, the index value is located from the bits 30 of the virtual address and is 9 bits in total. Then the function "find_pml4t_entry" is called to get the page directory pointer table and check if it returns NULL. If it returns a valid table, the correct entry is found in the table using the index value. If the present attribute is 1, then it means that the value in the entry points to the next level table which is the PD. If the present bit is cleared, then it's an unused entry and the value of "alloc" is checked. If it is equal to 1, a new page is allocated and the en-

try is set to make it point to this page. In the end the address of the page directory table is returned.

The next function is "find_pml4t_entry" which has the same parameters as "find_pdpt_entry". The PML4 table holds the entries whihc point to the page directory pointer tables. So the variable "map_entry" is defined which is a pointer to type "pdptr" and variable "pdptr". Since the entry in PML4 table is found, the index is locate from bit 39 of the virtual address. Then the correct entry is located and checked to see if the present bit is set. If it is set, the address is copied to "pdpt", otherwise the "alloc" is checked. If it is 1, a new page will be allocated and the entry is set to point to the page. At the end "pdptr" is returned.

The last function is "switch_vm". With the translation table prepared, the cr3 register is loaded with the new PML4 table. The virtual address is converted to the physical address because the cr3 register stores the physical address of the table.

25 Free memory pages

Now the system needs to free memory. Suppose we want to free a page, the entry in the page directory table is first located and the page is freed using the physical address saved in the entry. When the whole memory needs to be freed, the physical pages will be freed first, then the page directory tables, page directory pointer tables and as last the page map level 4 table. Generally, there could be a few page directory pointer tables and page directory tables so it is looped through each of the tables in the code. This will be implemented in the C file "memory.c".

25.1 Implementation

The first function is "free_pages". This is a bit the reverse process of mapping pages. The index is used to locate the correct entry in the page directory table. When the function is called, it is assumed that the "vstart" and "vend" are page aligned. The page directory table is found. Notice that the "alloc" variable is specified with value 0 which means it returns NULL if the entry does not exist. Because we want to free an existing page. If the return table is valid, the index is used to find the corresponding entry. Because the table in this case is the page directory table, the entry in the page directory table is pointing to 2 MB physical page. The lower 21 bits should be cleared before the address is used. Then the physical address is converted to a virtual address and the function "kfree" is called to free the page. After that, the entry is cleared in the table indicating that the entry is now unused. Then the system moves to the next page by adding the page size and continue the process until the end of the memory region is reached.

The next function is called "free_vm". When a process is build, a vm is created and reed when the process is exited. To free the vm, the physical pages will be freed in the user space as well as the page translation tables. because page translation tables are needed to locate the physical pages, the pages will first be freed by calling the function "free_pages". Then the page directory tables and page directory pointer tables are freed. The last one is the page map level 4 table. So the higher level page tables are freed after the lower level ones because they are needed to locate the lower level tables.

The function "free_pdt" will loop through the PML4 table and the page directory pointer tables to find the page directory tables. A variable "map_entry" is defined pointing to the PML4 table. Since each entry in the PML4 table points to the page directory pointer table, there could be a total of 512 PDP tables. If the entry is present, the address of the entry will be retrieved. This address is the address of the page directory pointer table. The PDP table also includes 512 entries which points to the page directory tables. So we loop through the entries as well. If this is a valid entry, the page is just freed and the entry is cleared meaning that this entry is not used. When returned, the page directory tables are freed.

The "free_pml4t" function is simple because the function "kfree" is called to free the PML4 table.

26 User Space

The last thing that is done in this part is creating a virtual space for user applications. In the system, the base of the virtual memory for user space is 0x400000 and only one page is mapped for the user programs which means the code, data and stack of the programs are located in the same 2 MB page.

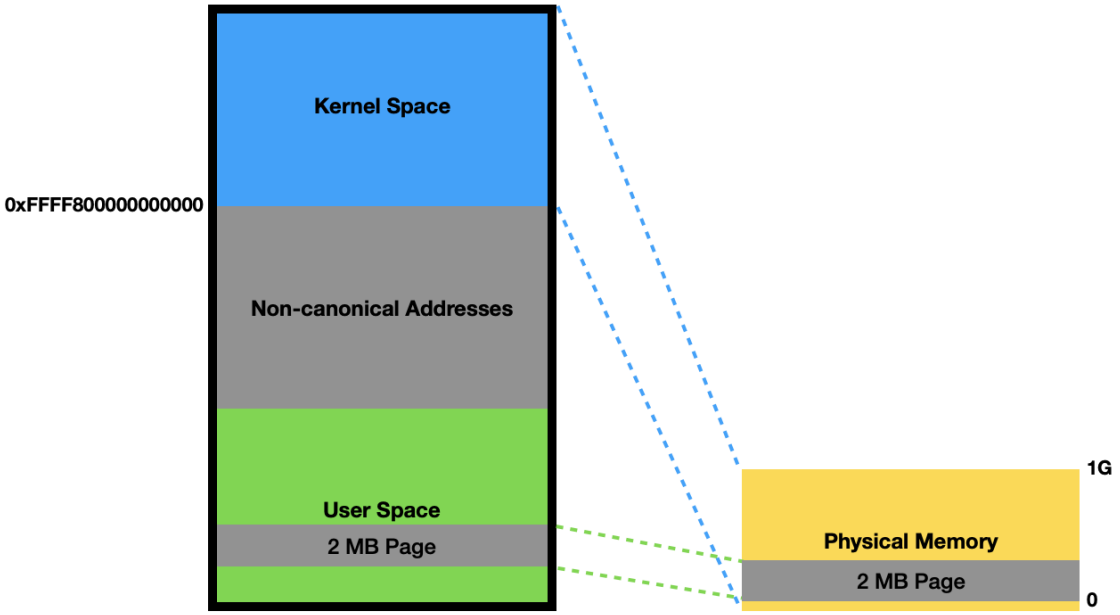

Figure 38: Page mapping

26.1 Implementation

The "setup_uvm" function is implemented in the "memory.c" file. The function takes three parameters, the address of the PML4 table, the location of the program and the size of the data that is copied. After allocating a page and mapping virtual address 0x400000 to this page, the program instructions and data are copied to it. The size references to the size of this data. A page is allocated which is used to store the code and data of the program. So the function "kalloc" is called. If a page is allocated successfully, the page will be zeroed using the function "memset" because it could include random values. The next thing that is done is mapping the page using the function "map_pages". The first argument that is passed to this function is the address of the PML4 table, so the "map" variable. The next two arguments are the start and end address of the virtual space we want to map (0x400000 in this case). Because only one page is implemented for a user application, the page size is added to the base address. Next parameter is the base of the physical page we want to map into which is the page that is allocated. The last parameter stores the attribute of the page. If this function succeeds, the data is copied to the page with the function "memcpy". if it fails, the page will be freed as well as the translation tables. At the end of the function, the status is returned.

Note that the kernel pages, where the kernel resides, are not freed because in the system there can be multiple vms and each vm is mapped to the same kernel pages. The kernel is residing in the memory until the system is shut down. Therefore, when a vm is freed, the kernel page is not freed because the kernel page is shared among all the vms.

Part IX
Processes

27 Details

27.1 Process

A process is basically a program in execution. It contains program instructions, the data the program needs, heap and stack within the address space. Each process has its own address space and can be executed by one or more threads. While a computer program is a passive collection of instructions typically stored in a file on disk, a process is the execution of those instructions after being loaded from the disk into memory. The operating system kernel resides in the kernel space of each process.

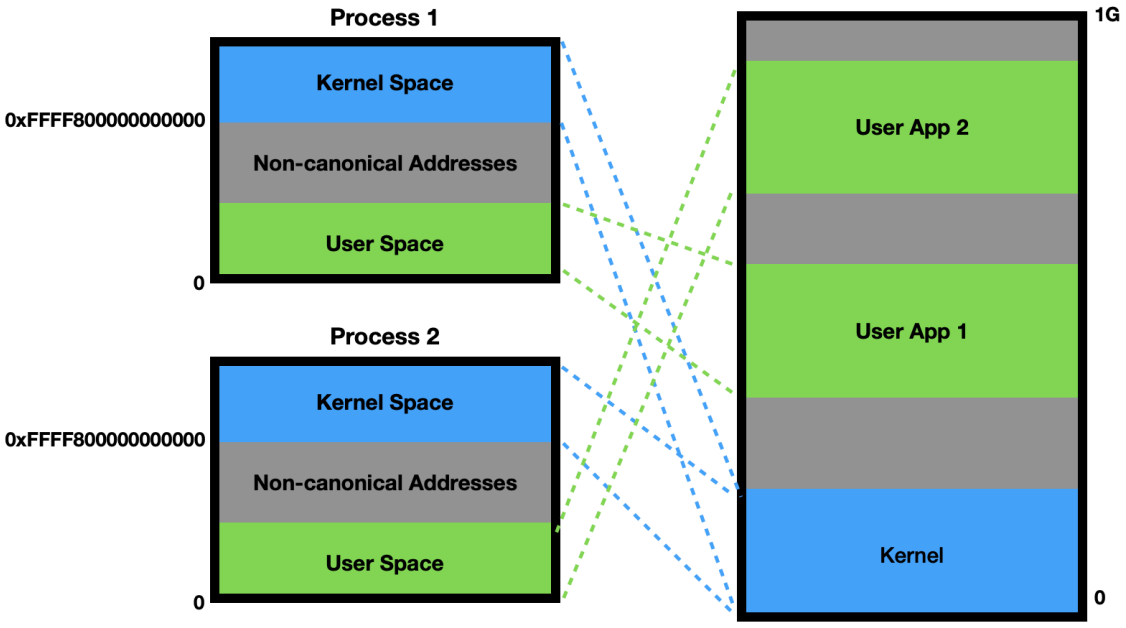

Figure 39: Mapping of processes

Because the operating system kernel resides in the kernel space of each process, the kernel space is mapped to the same physical pages where the kernel is located. The kernel space is thus the same among all processes. However, the user spaces of these two processes will map to different physical pages. So the process user space data saves its own program instructions and data.

27.2 Process control block

To store all the data of a process, a process control block (PCB) is used by the operating system and a user program is not allowed to access it. When a process is created (initialized or installed), the operating system creates a corresponding process control block. The role of the process control blocks is central in process management. They are accessed and/or modified by most utilities, particularly those involved with scheduling and resource management.

28 Initialization

The logic of processes are implemented in the ”process.h” and ”process.c” files.

28.1 The structures defined

In the header file the process structure is defined which is like a process control block. In this structure the following variables are stored. The identification number of a process. The state which indicates the status of the process (such as new, running, waiting, ready and terminated). The page map saves the address op page map level 4, when we run the process, we will switch to the current vm. The stack is used for the kernel code, a process has two stacks, one is used for user mode and another is for kernel mode. The stack here is used when kernel mode is entered. The last one is the trap frame. The reason that a trap frame is need is because in the system, we have two entry points when we switch from ring3 to ring0. One is through interrupts and the other is through exceptions. but since they are handled in the same function handler, this is actually the only entry point.
Next is the TSS structure, this structure is only for setting up stack pointer for ring0.
There are also some constants defined such as the stack size which is 2 megabyte, the number of processes is set to 10 so there could be a total of 10 processes running in the system. The next are the states a process can have.

28.2 Initialization of a process

The "process.c" file implements the functionality so that each process is initialized and run properly. There is a function to initialize a process (called "init_process"), where the system will try to find an unused process slot in the process table. After we find a process, the next thing that is done is setting up the process structure. Here, members of the structure are set properly.

29 System call

A system call is a request of a user program to the operating system to execute a specific task for the program. All system calls together form the interface (the API) of the operating system or kernel. The system calls are going to be used to print messages in the user programs and do other tasks.

Setting up the system calls is pretty much the same as we implemented the hardware interrupts. The major difference is that we use the instruction INT to fire the software interrupt. For example, the first thing that is implemented is the print function to let a user program print characters.

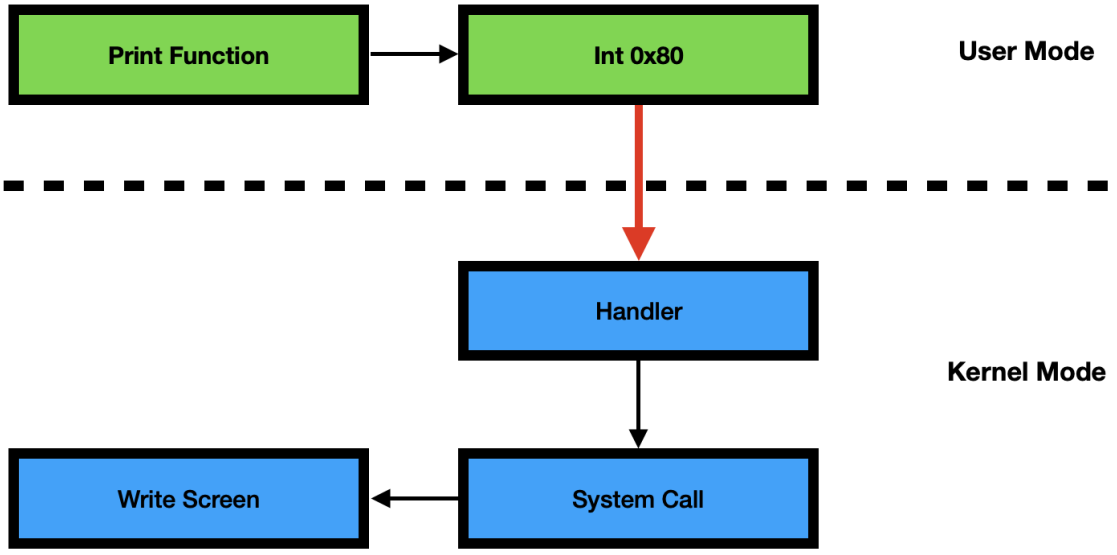

Figure 40: Print function

The print function that is implemented in the operating system, will execute an INT instruction with vector number 80 (0x80). This is the part that will be done in user mode. When running the INT instruction, the CPU will retrieve the corresponding descriptor in the IDT (see part V: Exceptions & Interrupt Handling). Then kernel mode is entered and this request is processed in the handler. After it performs some checks of the request, it gets to the system call part and the write screen function will be called. Then the message will be seen on the screen.

29.1 Kernel mode of system call

Because the system needs to make use of a vector (0x80), it has to be initialized the same way as we initialized vectors. Note that this is an INT instruction so the initialization is a little bit different. It is thus written in the "trap.h" file with all the other interrupt vectors. The entry point for this interupt is sysint and the attribute of the IDT entry is different from others. Here it is set to 0xee. The difference is that DPL is set to 3 instead of 0 because the interrupt will be fired in ring 3. Otherwise we have no privilege to access this descriptor. The declaration of the function is also added in the header file "trap.h" like the other interrupt vectors. The entry point "sysint" is also declared in the "trap.asm" file. 0 is pushed which means it includes no error code. Then trap number 80 is pushed so that we know this is the software interrupt in the handler function.

In the handler function in file "trap.c", the switch statement is changed with an extra case (0x80). When this case is called, the "system_call" function is called with a reference to the data in the user stack.

The "sys_call" function is declared in the "syscall.c" file. The array "system_calls" holds the system call function pointers. The number of the function that can be used is 10 which is enough. The function "init_system_call" simply saves the system write function in the first element of the array. The "system_write" function is the function that is used in user mode. To find the correct system call, the function "system_call" is implemented.

Now to the user program. Note that we cannot use the C standard library when the user program is written. So the first thing that is done is writing a simple library for the user program. A "print.c" file is added in the library folder, this folder can be used by any user. The "printf" function is implemented which is not any different from the "printk" function but here functions are called to send requests to the kernel.

30 Scheduling

To switch between processes, the systems needs a scheduling algorithm in the process module. The scheduling algorithm that is used in the operating system is round robin. This is the easiest scheduling algorithm for a preemptive scheduler. Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in the ready queue till it gets its next chance to execute.

Each process is assigned a time slice or "quantum". This quantum dictates the number of system timer ticks the process may run for before being preempted. To achieve this, the running process is given a variable that starts at its quantum, and is then decremented each tick until it reaches zero. Suppose there are 3 processes and after we initialize the processes, we call the function to launch the first process. From now on we are in user mode, so how do we force the process to give up the CPU resource and pick another process to run. There is already a timer interrupt implemented which is configured to fire interrupts every 10ms. In user mode, the interrupt is enabled so the timer interrupt will be processed by the processor and the control is transferred to the trap handler.

In the handler is checked to see if this is the timer interrupt. If it is the timer interrupt, we

will go to the process module to do the context switch and then the second process is selected. When we jump back to user, the second process is actually run. After it runs for a period of time (10ms in this case) the timer interrupt will be fired and the system gets to the trap handler again. Choose the third process and we get to user mode to run the program. When the time for this process is up, we will select the first process in the handler again. In the system, the switching among different processes is continuously done.

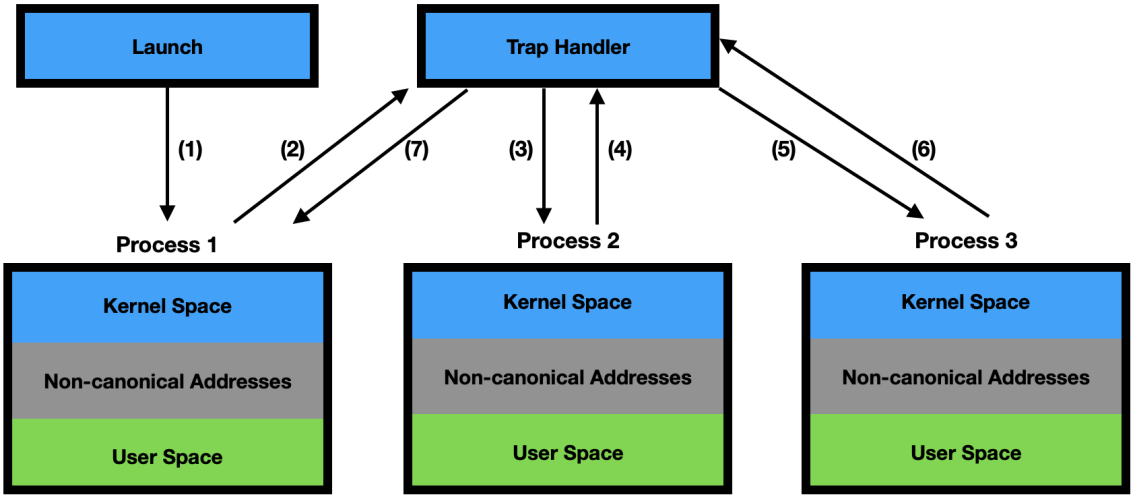

Figure 41: Sequence of processes

In the process module, the runnable process is saved in the linked list called ready list. Suppose the 3 processes are in the ready list and the first one is selected. The first process goes out the ready list because it is running and there are now 2 processes in the list. The content of process 1 will then be restored and running until the next timer interrupt occurs. When the interrupt is fired, the process 2 is removed from the list and process 1 is appended to the ready list. Then process 2 will be run.
Since the runnable processes are saved using a linked list,the member list is added to the process structure in the "process.h" file. The structure of the list itself is implemented in the library module "lib.h" file.

30.1 Implementation

First off is the "lib.h" file. Since we save the runnable processes using a linked list, the member list is added in the process structure. The structure list is implemented in the library module ("lib.h"). the structure "List" includes only one item which is the pointer to the next list structure. The structure "HeadList" contains the head and tail of the list. The field next points to the process which is supposed to be running next and the current process is added at the tail of the list.

In the C file "lib.c" are three functions. Append an item to the list, remove an item from the list and check the list status. These functions are fairly simple so aren't explained in this paper.

Now, new functionality is added in the process module to do the context switch. In the "process.c" file, the initialize process function is changed. A for loop is used to find the unused process and set the process entry.

31 Sleep and Wake Up

A blocking process is usually waiting for an event such as a semaphore being released or a message arriving in its message queue. A special case of waiting is sleeping, which is when a process is

set inactive for a given period of time. This is usually handled separately for ordinary waiting, primarily for efficiency reasons; since the clock interrupt is frequent, and keeping track of individual ticks is infeasible, the usual approach is to use a system of relative counters to mark when a given process will be awakened. So there also has to be a functionality to shift the state of the process between sleep and active which can be done by scheduling. When the handler is called from a process, that process can go into the sleep state. Then the scheduler is called to select another process. The scheduler will select all processes after a certain time unless a process is set into the sleep state. This process will not run until it is set active again. It can be set active again by waking up the process.

31.1 Implementation

The sleep and wake up functionality is implemented in "the process.c" file. Notice that there is a wait parameter in both functions. This is because the process can be in sleep state or wake up state for various reasons. This wait parameter is added to the process structure and will give the info as to why the process is put into sleep and how to wake it up.
In the "sleep" function, the first thing that is done is collect the data of the current process which is going to be in the sleep state. Then the state of the process is put into the sleep state and the reason is put in the wait field. Then the process is appended into the wait list. This is commonly called the delta queue, which saves the process and for how long it is sleeping. Because the wait time is implemented in the process structure and given to the wait list, the wait list will not have to store the wait time separately. Then the scheduler is called to select another process.
The "wake up" function is going to find the correct process in the wait list and remove it, then append it to the ready list waiting for scheduling. Generally, there could be multiple processes waiting on the same object. To check all. these processes a while loop is implemented used to find all the waiting processes in the list. If one is found, the state will be changed to ready again and appended to the ready list.

32 Exit and Wait

A program also needs to be exited at a certain time. In this system, it will be done with the function exit and wait. When a program exits the function exit is called and the system call function sys exit will receive this request. Then it will append the process to the killed list and then switch to other processes. When the init process function gets run, it will call the function wait to do the cleanup. So here the initialized process is the process responsible for cleaning up the other processes when they exit.

32.1 Implementation

Everything is again implemented in the "process.c" file for the exit functionality. As in the functionality of sleep and wait, the info of the current process is retrieved and the state of the process is changed to exit. The process will then be appended to the killed list. Then the wake up function is called because the initialization process, which is the first process in the system, is constantly calling the wait function to do the cleaning up and it could be in sleep state. So it has to be woken up before we switch processes. The value of the parameter is 1 because we want to wake up the process with process id 1 (which is the first process). At last, the function will jump to the scheduler.
The wait function will retrieve the killed list and loop infinitely to constantly find exit processes and do the cleanup.

33 Terminate a process

The handler in the "trap.c" file will halt the system if an exception or unknown interrupt occurs. But now there is an exit and wait function in the kernel so the system can force a process to exit if it causes exceptions or unknown interrupts.

To achieve this, there will first be checked in the handler function (”trap.c”) if the exception is generated in user mode. If the lower two bits are equal to 3, then it is in user mode. If that is the case, this is the error in the program and we call the function exit to kill the process an never return. When the wait function is called the process will be freed. A message is printed so that we can see which exception it generates. If the exception is generated in kernel mode, the system is halted.

Part X
PS/2 Keyboard Driver

To interact with the kernel in the console, a keyboard is added. The type of keyboard we use is a PS/2 keyboard.

34 Details

34.1 PS/2 port

A PS/2 keyboard is a keyboard that uses the PS/2 port and is a 6-pin mini-DIN connector used for connecting keyboards and mice to a PC compatible computer system. This port is outdated but is one of the easiest to write code for.

34.2 Device driver

To let the keyboard interact with the kernel, a driver is needed. a device driver is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details about the hardware being used.
A driver communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device (drives it). Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

35 The driver implemented

35.1 Interrupt request

The keyboard is connected to the PIC using irq1. So the keyboard uses interrupt signal 1, and should be enabled on the PIC chip so that the keyboard interrupt will be sent to the CPU. Generally, the keyboard should be initialized before it is used, but in the operating system it is assumed that the initialization of the keyboard is done before the kernel is running. The PIC configuration is done in the kernel file ”kernel.asm”. In the previous part (Part IX: Processes) all the interrupts where masked except the timer interrupt. Now the second bit, which is IRQ1, is unmasked (setting bit from ’1’ to ’0’).

The next thing is installing the keyboard interrupt service routine so that the handler will be called when the keyboard interrupt is fired. This is done in the ”trap.asm” file, the vector 33 is defined. First zero is pushed which means that there is no error code. Then index number 33 is pushed and jumped to trap. Note that the vector is globally declared so that it can be referenced in other files. The vector is also defined in the corresponding C files ”trap.h” and ”trap.c” so that

we receive the keyboard requests in the interrupt handler. If the index number is 33 in the handler function, the keyboard handler can be called to process this request.

35.2 Keyboard handler

Everything necessary for the keyboard is available in the "keyboard.c" file. There are some arrays defined at the top of the file. These arrays are key maps that represent a character on the keyboard. Note that the data sent form the keyboard to the handler is not the data in these arrays. The data in the key maps are called scan codes, which means we need to convert the scan code to ASCII code for the keyboard character and print it on the screen. The scan code is sent to the handler when the user presses a key and releases a key. There are two scenarios key up and key down or make code and break code. The scan codes can be found in [4]

There are 3 sets of scan code but the keyboard driver in this operating system works with scan code set 1. We have one byte for a key, multiple-byte code used for function keys which will be sent one byte at a time. Most of the multiple-byte key code comes with E0 first. There is a separate key map used for when the user holds the shift key or presses the caps lock key. Notice that the driver doesn't support the numeric keypad and function keys except the caps lock and shift keys.

The function "keyboard_read" reads a key from the keyboard. To read a scan code, we read from port 60. If the scan code starts with e0, the e0 sign is added to the flag meaning that the key is not valid (because this driver only uses one byte for a key). The rest of the code handles the special cases (shift and caps lock) and converts the scan code to ASCII code.

In the "keyboard_handler" function, the character is simply printed on the screen if the return value is larger than 0. Note that the array for the character is defined with 2 elements. The reason we define the array that way is because the print function doesn't support printing single characters so instead it is saved as a string with the second element being the null character.

There is still one special case, that is, when we press backspace to delete characters which is handled in the "print.c" file. There the current position of the cursor will be checked. If it is at the beginning of the screen nothing will happen and the loop is continued. Otherwise, we will check the column. If the cursor is at the start of the line, the row is decremented and change the column to the end of the previous line.

35.3 Key buffer

A key buffer is the last thing that is added so that the other modules and user programs can read the keys instead of just printing it on the screen. To implement this we, the already implemented keyboard handler will be used to receive the key characters and all that needs to be done is put the characters in the buffer. The buffer is defined in the "keyboard.h" file and will be used to manage the read and write operation of the key buffer. The buffer is an array with 500 bytes, there is also a front and end which specifies the current front and end of the characters.

Now in the "keyboard_handler" function if a character is read from the keyboard and it's a valid character, it is written in the buffer. There is checked that the buffer isn't full. The buffer is a circular queue, so if the end is one element away reaching to the front, it means that the buffer is full and we simply return. Otherwise we write the data in the buffer, update the end and write it back to the key buffer for later use. If the element is on the last position of the buffer, then the following element is going to be placed back at the start. After the writing of the character, the operating system wakes up the process waiting for the keyboard. In this system, when a program wants to read a key and there is no key in the buffer, we will put it into sleep. The wait argument that is passed in this case is -2 which represents the processes waiting for the keyboard IO.

Next up is reading the buffer. This is done in the "read_key_buffer" function. In this function it is first checked if the buffer is empty before we read a key. If it is empty the system will put

the process into sleep and if we have keys in the buffer the key will be returned to the caller and update the front to point to the next location.

36 Interact with the kernel using the console

The last thing implemented in the system is that a user can write to the console where it can issue a command and get the result from the kernel. The commands are going to be implemented in a user program because a user will use the commands.

36.1 Implementation

The code for the commands is added in user 2 in the C file "main.c". In the main function variable "buffer" is used to save the data the user sends with the keyboard. The buffer size specifies how many characters are gotten in the console before a command is issued. Each time enter is pressed, the buffer size is reset to 0. The "cmd" variable is the index of the commands that are supported. Then there is an infinite loop where data is received, a command is issued and the result is printed on the screen. The layout of the screen is as follows. At the beginning of each line, the command prompt "shell#" is printed. Then the command that the user wants to execute is typed. Enter is the pressed to issue the command. Next, the data from the keyboard will be read. When the function returns, the buffer size is checked. If the buffer size is 0, then there is no command found and the loop is continued. Otherwise, the commands are parsed. If the return value is negative, it means that the command is invalid and the prompt "Command Not Found!" is printed. Otherwise, this is a command the system supports. The command is executed by the function "execute_cmd".

The function "read_cmd" will read keys from the keyboard. The else clause deals with the normal keys. In this case, the data is copied to the buffer and the buffer size is updated. Then it is printed on the screen. If it is an enter key which means the user wants to issue a command. The enter key is printed, the loop is breaked and the buffer size is returned. The command size is capped on 80 characters. Another special case is when the backspace key is pressed to delete characters. If the buffer size is 0, it means that the user is at the beginning on the line and nothing can be deleted. So nothing is done. Otherwise the buffer size is decremented and the backspace is printed.

Next function is the function "parse_cmd". First off, a variable "cmd" is used as an index number. It is initialized with -1 which means that it is an invalid index. If an if statement passes, then the appropriate index is given. After the index number is received, the command can be executed using the function "execute_cmd".

In this function a command list is created with all the possible commands that the user can access. Each command will call a different function that in its turn will send a request to the kernel.

Part XI
Sources

References

[1] *Bochs installation.* URL: https://bochs.sourceforge.io.

[2] *Convert to assembly using nasm.* URL: https://www.ele.uva.es/~jesman/BigSeti/seti1/nasm/nasmdoc2.html.

[3] *Creating a disk image.* URL: https://www.netadmintools.com/art476.html.

[4] *Scan Codes Set 1.* URL: http://users.utcluj.ro/~baruch/sie/labor/PS2/Scan_Codes_Set_1.htm.