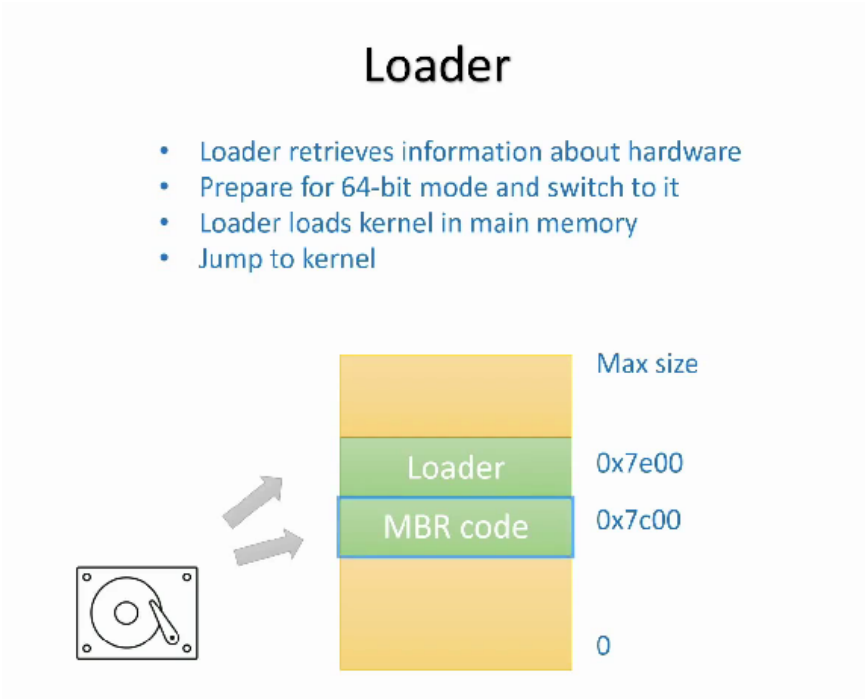
Loading the Loader and Switching to Long Mode

Once the disk extension check is successfully passed, we proceed to load a new file called "loader" into memory. The reason we require a separate loader file is that the Master Boot Record (MBR) has a fixed size of 512 bytes. Within this limited space, there are reserved areas for other purposes, such as partition entries. As a result, we have less than 512 bytes available for the boot code itself.

In the boot process, there are several tasks that need to be accomplished, including loading kernel files, obtaining the memory map, switching to protected mode, and jumping to the long code. However, fulfilling all these tasks within the 512-byte limit of the MBR is not feasible. Therefore, we utilize a separate loader file to perform these tasks, as it does not have the same size restrictions.

By using the loader file, we can extend the capabilities of the boot process beyond the limitations imposed by the MBR. This allows us to load the necessary components, retrieve memory information, transition to protected mode, and execute the desired code seamlessly. The loader file provides the necessary space and flexibility to accomplish these tasks effectively, ensuring a smooth and efficient boot process.



To store the loader file immediately after the location where the boot code is stored, which is at memory address 0x7e00, we define a label called "LoadLoader" in our code. In order to access the disk and load the loader file, we utilize interrupt 13, with function code 42. This function code is stored in the ah register, indicating our intention to use the disk extension service.

Before invoking the disk extension service, it is essential to ensure that we provide the appropriate drive identification. We store the drive ID in the dl register, specifying the specific disk drive from which the loader file should be loaded. This step is crucial to accurately access the desired disk and retrieve the required file.

By setting up the necessary parameters and calling interrupt 13 with the appropriate function code and drive ID, we initiate the process of loading the loader file into memory. The loader file will be stored immediately after the location of the boot code at memory address 0x7e00, enabling us to proceed with the subsequent tasks in the boot process seamlessly.

```
mov dl,[DriveId]
mov ah,0x42
```

To pass the necessary parameters to the disk extension service, we utilize a structure called "Read Packet" with a size of 16 bytes. This structure serves as a container for the specific information required by the service to perform the desired disk read operation.

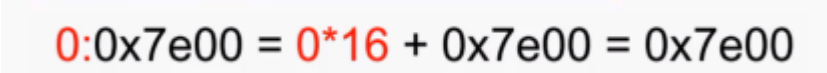
The exact contents and layout of the Read Packet structure may vary depending on the specific disk extension service and the platform being used. However, in our case, the structure contains the necessary parameters for reading data from the disk.

The Read Packet structure allows us to specify details such as the starting sector or block from which to read data, the number of sectors or blocks to be read, and the memory location where the data should be stored.

By organizing these parameters within the Read Packet structure, we can pass the entire structure as a parameter to the disk extension service, ensuring that the service has access to all the required information to perform the read operation correctly.

```
mov si,ReadPacket //now si hold the memory of Read Packet
```


```
mov word[si],0x10
mov word[si+2],5 //5 is enough here
//7e00 is stored to first word
mov word[si+4],0x7e00
mov word[si+6],0
```




```
//The last two words are 64bit logical block address
//Loader file will be written into the second sector of the disk
//lba is zero-based address that the first sector is sector 0, the second sector is se
ctor 1 and so on.
mov dword[si+8],1
mov dword[si+0xc],0
int 0x13
jc ReadError
```

To initiate the disk extension service and perform the required disk read operation, we call interrupt 13. Before making the call, we ensure that we pass the appropriate drive ID to the loader. The drive ID specifies the specific disk from which we want to load the kernel file.

In the event that the disk extension service is not supported by the system, an error occurs. To handle this situation, we modify the error message to indicate that there is an error in the boot process. This message serves as a notification that the disk extension service is not available, and further actions may be required to resolve the issue.

It is crucial to include the drive ID when passing parameters to the loader. The drive ID specifies the disk from which the kernel file should be loaded, ensuring that the correct disk is accessed during the disk read operation. By providing the drive ID as a parameter, we can guarantee that the loader fetches the kernel file from the intended disk, allowing the boot process to proceed correctly.

```
//in Load Loader:
mov dl,[DriveId]
jmp 0x7e00 //which is the address of memory of which we load our loader from disk
```

To load the "loader.asm" file in our project, we make modifications to the "build.sh" script. Within the script, we add the necessary code to facilitate the loading process.

The code added to "build.sh" is responsible for handling the loading of the "loader.asm" file. This file contains a print message that we want to include in our project. By adding the code in "build.sh," we ensure that the "loader.asm" file is processed and integrated into the build process.

This addition to the script allows us to incorporate the functionality and content of the "loader.asm" file into our project, enabling the execution of the print message contained within it during the boot process.

```
nasm -f bin -o loader.bin loader.asm //this command is also added along with boot.asm
 one
//then we use dd command
dd if=loader.bin of=boot.img bs=512 count=5 seek=1 conv=notrunc //here we use count=5
 to write 5 sectors of data in boot image
```

In our boot image, the first sector is reserved for the storage of the "boot.bin" file. This file contains the initial boot code that is executed when the system starts up. The "boot.bin" file is responsible for loading the subsequent code and initiating the boot process.

After the first sector, we write the contents of the "loader" file starting from the second sector. By using the "seek=1" parameter, we instruct the system to skip the first sector when writing the "loader" file. This ensures that the "loader" code is stored in the appropriate location within the boot image, following the "boot.bin" file.

By organizing the boot image in this manner, we separate the initial boot code and the subsequent loader code, allowing for a structured and organized boot process.

The "boot.bin" file sets the foundation for booting, while the "loader" file builds upon it to perform additional tasks and functionalities during the boot sequence.

```
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o boot.bin boot.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o boot.bin /mnt/c/Users/viswa/Desktop/boot/boot3.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o loader.bin /mnt/c/Users/viswa/Desktop/boot/loader.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=boot.bin of=boot.img bs=512 count=1 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.0035634 s, 144 kB/s
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=loader.bin of=boot.img bs=512 count=5 seek=1 conv=notrunc
0+1 records in
0+1 records out
33 bytes copied, 0.00334388 s, 9.9 kB/s
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$
```

```
loader starts (PCI) 0.8a 03 Jun 2021
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs 2.7 BIOS - build: 08/01/21
$Revision: 14314 $ $Date: 2021-07-14 18:10:19 +0200 (Wed, 14 Jul 2021) $
Options: apmbios pcibios pnpbios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk (    9 MBytes)

Press F12 for boot menu.

Booting from Hard Disk...
```

Load Mode Checker

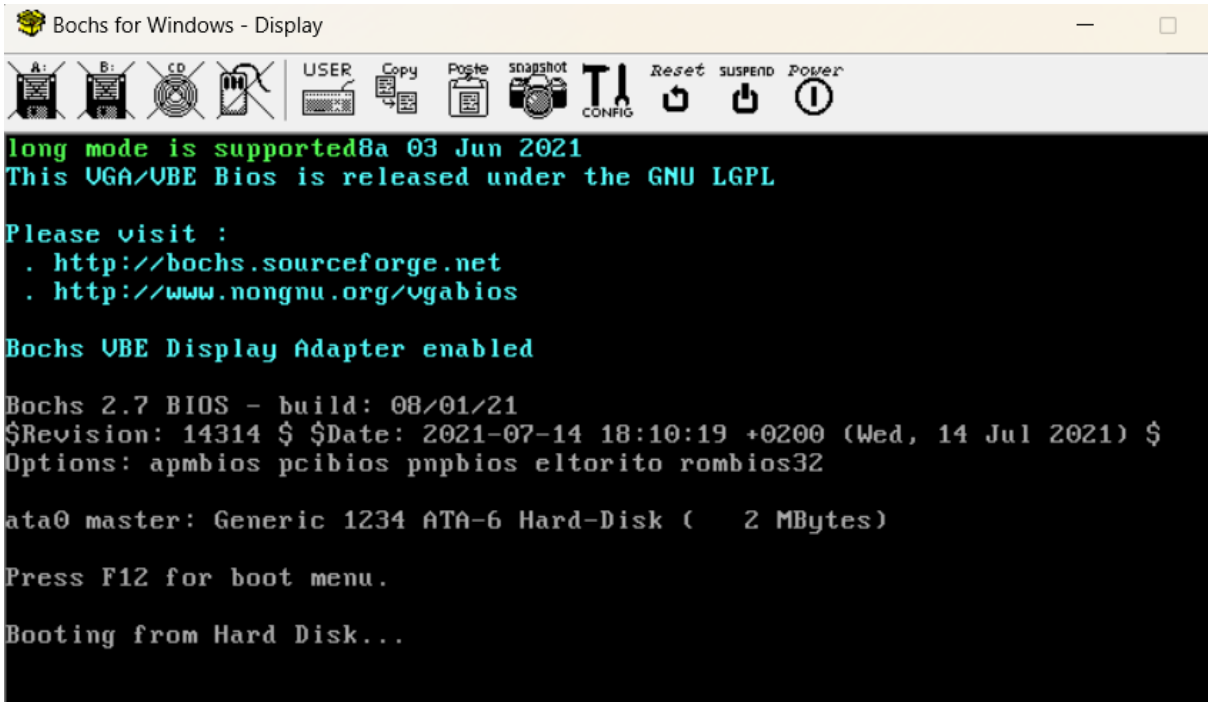
To determine if our CPU supports long mode and 1G page support, we utilize the CPUID instruction. This instruction provides processor identification and feature information. We pass the value 0x80000001 to the EAX register and execute the CPUID instruction. By varying the value in EAX, we can retrieve different information about the processor.

The information regarding long mode support is stored in the EDX register. We specifically check bit 29 in the EDX register. If this bit is set, it indicates that the processor supports long mode. Conversely, if the bit is not set, it signifies that long mode is not supported. We can determine the status of the bit by checking the zero flag after performing the test.

Additionally, we also check if 1G page support is present. This feature indicates whether the CPU supports the use of 1G pages for memory management. Similar to checking long mode support, we examine the relevant bit (bit 26) in the EDX register.

If the bit is set, it indicates that the CPU supports 1G page support. If the bit is not set, it indicates that this feature is not supported.

To perform these checks, we assign the value 0x80000000 to the EAX register before executing the CPUID instruction. This allows us to retrieve the relevant information about the processor's features and capabilities.



Load Kernal File

The next step in our project is to load our kernel file. The process is similar to that of the loader file, but with some necessary parameter changes. We create a label called "LoadKernel" to handle this task. In this case, we choose to load the kernel at the address 10000.

To specify the destination address, we need to store the values for offset and segment. Initially, we may be tempted to directly move the value 10000 into the memory location pointed to by the SI register. However, this approach would result in an overflow error since the memory location only has a word size (16 bits).

To handle this correctly, we follow a different approach. We first move the value 0 to the memory location at SI+4, which represents the offset. Then, we move the value 1000 (segment value) to the memory location at SI+6. This ensures that the address is stored correctly without encountering an overflow issue.

Following the loading process, we place a label called "read_error" at the end of the code. This label is called if the kernel fails to load successfully. Conversely, if the

loading process proceeds without errors, we print the message "The kernel is loaded."

By implementing these steps, we can successfully load the kernel file into memory and proceed with the execution of our project.

```
mov word[si+4],0x10000 //this will result in overflow
//we should follow this way
mov word[si+4],0 //offset
mov word[si+6],0x1000 //segment
```

```
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o boot.bin boot.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o loader.bin loader.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=boot.bin of=boot.img bs=512 count=1 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.0143394 s, 35.7 kB/s
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=loader.bin of=boot.img bs=512 count=5 seek=1 conv=notrunc
0+1 records in
0+1 records out
147 bytes copied, 0.0143815 s, 10.2 kB/s
```

```
kernel is loadedCI) 0.8a 03 Jun 2021
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs 2.7 BIOS - build: 08/01/21
$Revision: 14314 $ $Date: 2021-07-14 18:10:19 +0200 (Wed, 14 Jul 2021) $
Options: apmbios pcibios pnpbios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk (   2 MBytes)

Press F12 for boot menu.

Booting from Hard Disk...
```

Memory Map

To obtain the memory map, we utilize the system map service through interrupt 15. This service returns a list of memory blocks, with each block consisting of 20 bytes. The memory block structure includes three offset fields:

- The first qword (offset 0) represents the 64-bit physical start address of the memory region.
- The second qword (offset 8) indicates the length of the region in bytes.
- The last dword (offset 16) signifies the type of memory, with a value of 1 denoting free memory that is available for use, while other values represent

different memory types.

In our project, our objective is to collect memory regions of type 1 only. By calling the system map service and iteratively examining each memory block, we filter and gather the relevant memory regions.

OFFSET	FIELD
0	Base Address
8	Length
16	Type

We present a detailed procedure for retrieving memory information using x86 assembly language. The process involves accessing the e820 service through interrupt 15 and sequentially retrieving memory blocks until the end is reached. We will outline the necessary steps and highlight key instructions and register usage.

Procedure:

1. Define Label: GetMemInfoStart
    - This label signifies the beginning of the memory information retrieval process.
2. Set Register Values:
    - Set the value of the EAX register to represent the e820 service.
    - Set the value of the EDX register to the ASCII code for "smap."
    - Set the value of the ECX register to 20, indicating the length of the memory block.
    - Clear the EBX register to 0 using the XOR instruction.
3. Save Memory Block Address:
    - Save the memory block address returned by the service in the EDI register.
    - This address will be used to access the memory block later.
4. Call the Service:
    - Use interrupt 15 to call the service and retrieve memory information.
    - Check the carry flag after the call to determine if the service is supported.
    - If the carry flag is set, jump to the "not supported" section.
5. Define Label: GetMemInfo

- This label indicates the continuation of memory information retrieval.

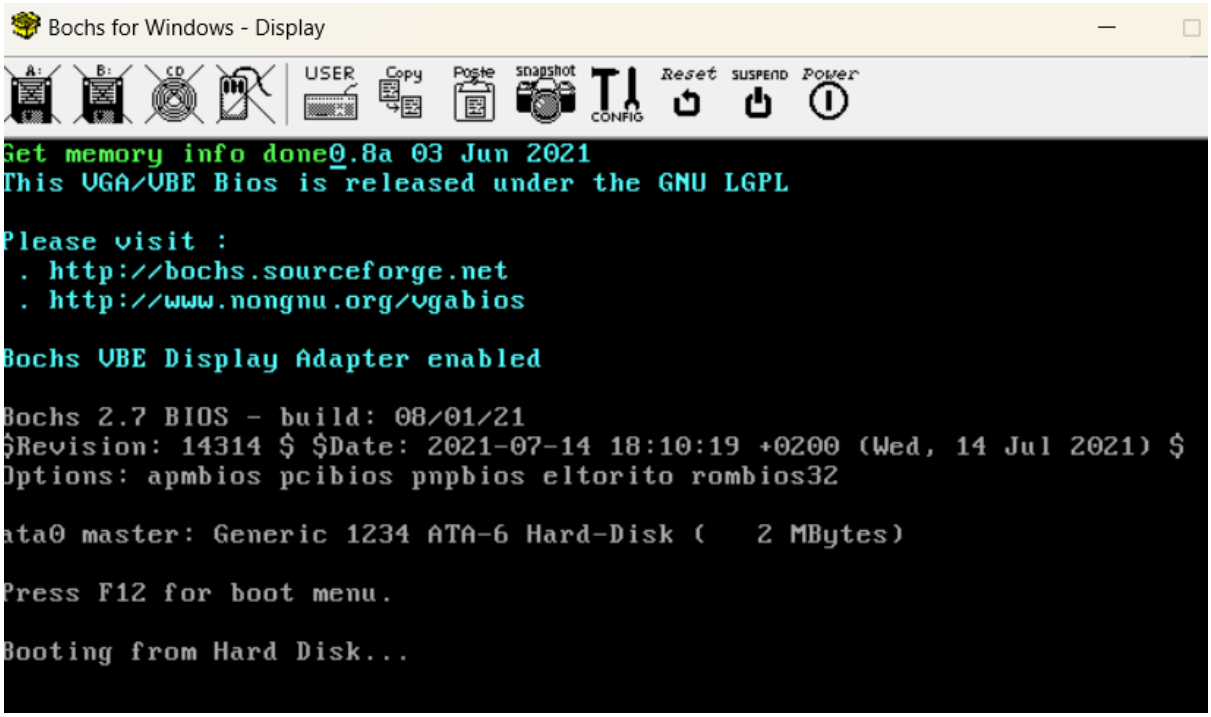
6. Adjust Memory Block Address:
    - Increase the value of the EDI register by 20.
    - This adjustment ensures the EDI register points to the next memory block.
7. Pass Parameters and Preserve EBX:
    - Set the registers EAX, ECX, and EDX with the same values as before.
    - Preserve the value of the EBX register for the next call.
8. Call the Service Again:
    - Use interrupt 15 to call the service and retrieve the next memory block.
    - Check the carry flag after the call to determine if it's the last memory block.
    - If the carry flag is set, jump to the "get memory done" section.
9. Define Label: GetMemDone
    - This label signifies the end of memory block retrieval.
10. Update Message:
    - Change the message to indicate that the memory information retrieval is complete.
11. Check EBX Value:
    - Test the value of the EBX register.
    - If EBX is non-zero, jump back to the "GetMemInfo" label.
12. Define Label: GetMemInfo
    - This label indicates the continuation of memory information retrieval.
13. Query Memory Info:
    - Continue querying the memory information by repeating steps 6-12.
14. Reach End of Memory Block:
    - If the end of the memory block is reached, the process is done.

The process of retrieving memory information in x86 assembly language involves setting the appropriate registers, calling the e820 service using interrupt 15, and sequentially accessing memory blocks. By checking the carry flag and the EBX

register, the program determines whether there is more memory information to retrieve or if the process is complete. This report outlines the step-by-step procedure for successfully retrieving memory information in assembly language.

Note: Here we just get the message that Memory map info is done since we haven't installed c compiler but in memory management session we will be looking at this in detail

```
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o boot.bin boot.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o loader.bin loader.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=boot.bin of=boot.img bs=512 count=1 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00821916 s, 62.3 kB/s
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=loader.bin of=boot.img bs=512 count=5 seek=1 conv=notrunc
0+1 records in
0+1 records out
217 bytes copied, 0.00464952 s, 46.7 kB/s
```



Test A20 Line

Back in the days when machines had 20-bit addresses, the maximum memory that could be addressed was 2 to the power of 20, which equates to 1MB. As technology advanced, newer machines were equipped with address buses wider than 20 bits. However, for compatibility purposes, the A20 line of the address bus was often disabled.

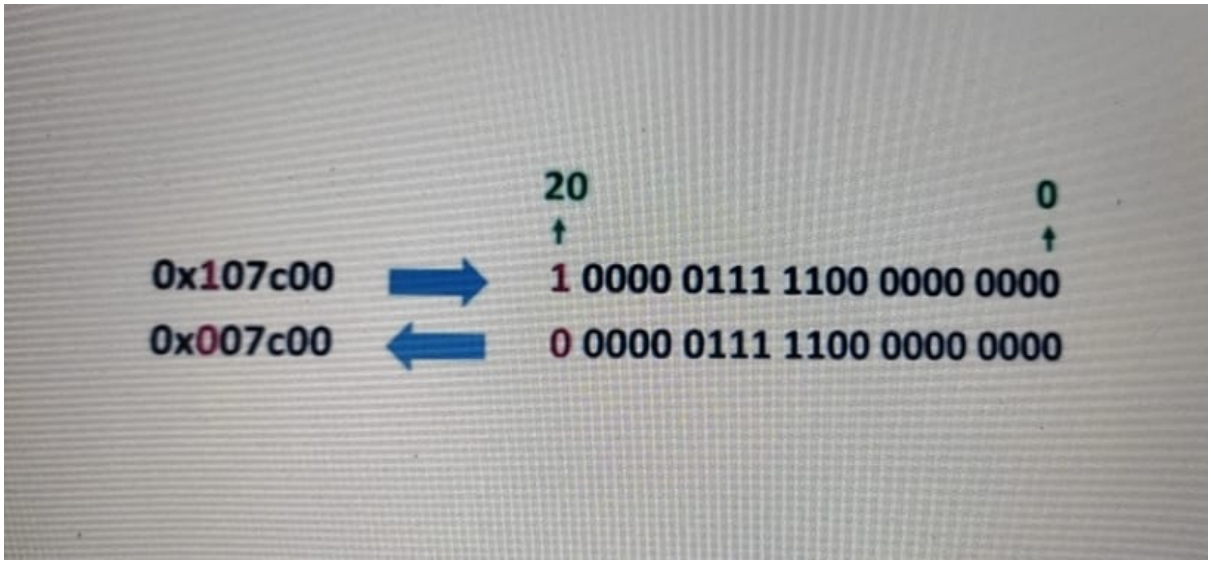
When the A20 line is off and an address higher than 1MB is accessed, the address is effectively truncated as if it starts from 0 again. This limitation means that only

even megabytes of memory can be accessed since the A20 line is 0, causing any address accessed to have bit 20 cleared. To overcome this restriction, the A20 line needs to be toggled to enable access to the entire memory range.

It is worth noting that different methods exist for activating the A20 line, and not all methods are guaranteed to work across all machines. However, in more recent machines, the A20 line is typically enabled by default. For the sake of simplicity in our project, we will assume that our machines have the A20 line enabled.

To proceed, we will test whether the A20 line is already enabled. If it is enabled, we will print a message indicating that the A20 line is on. Otherwise, we will halt the system and display a blank screen, as the A20 line needs to be enabled for proper memory access.

To test whether the A20 line is enabled in our project, we will employ a straightforward logic. If the A20 line is not enabled, attempting to access a specific address, such as 107C00, will reveal that bit 20 of the binary value is 0. Conversely, if the A20 line is enabled, bit 20 will be 1, resulting in the address being effectively truncated to 7C00.



In our project, we will utilize this method to perform the A20 line test. We have chosen the address 7C00 for this purpose, as it corresponds to the start address of the boot code. Since the boot code has completed its tasks, we can reuse this memory area for the test.

Let's proceed with the implementation. We define a label called "test_a20." To access memory at 107C00, we will move the value FFFF to the AX register and then

copy it to the ES register. Subsequently, we will write a random value, such as A200, to the address 7C00 using the DS register. By doing this, the logical address now points to 7C00, and the instruction will copy A200 to the memory at 7C00.

Next, we will compare the content at the address 107C00 with A200. To accomplish this, we will utilize the ES register as the segment register and set the logical address to ES:7C10. This logical address corresponds to the physical address 107C00, which is the target address for our test.

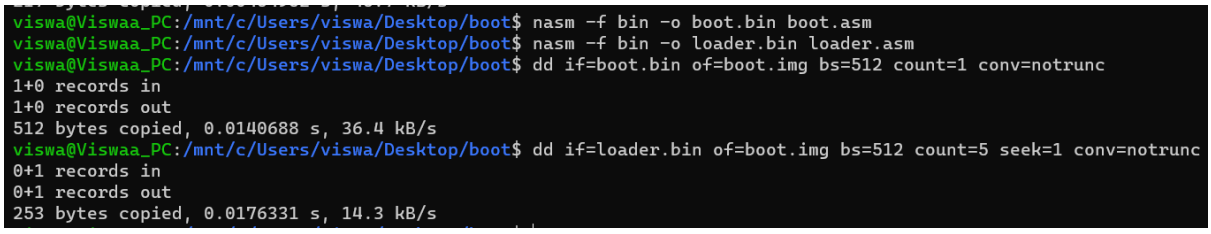
If the content at the address 107C00 is not equal to A200, we can conclude that we have successfully accessed the memory address 107C00, indicating that the A20 line is enabled. In this case, we will jump to the label "set_a20_line_done" and execute the code responsible for printing the message "A20 line is on."

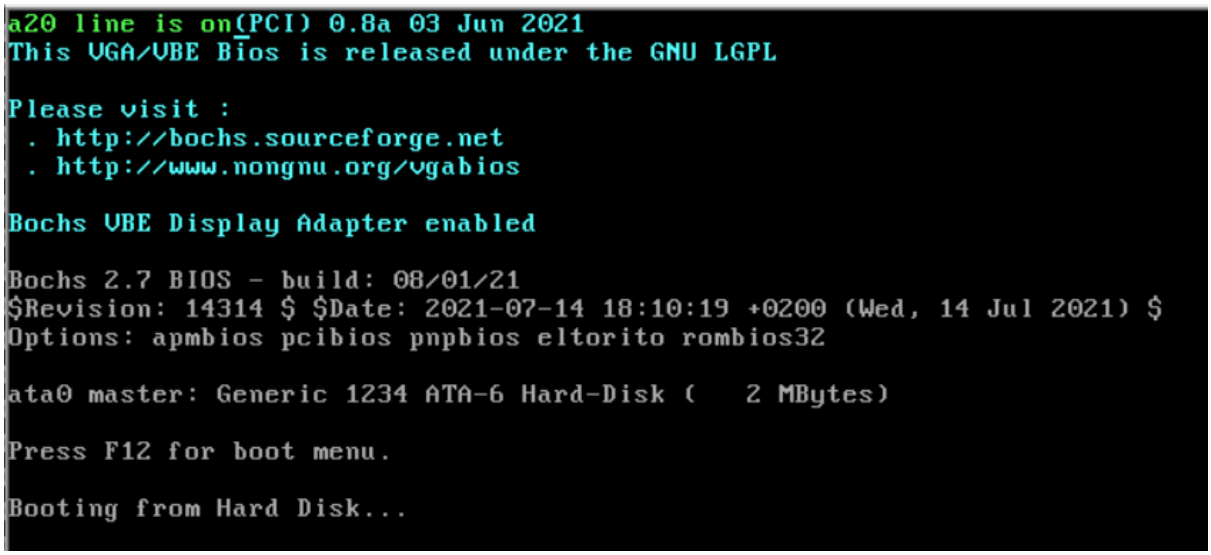
However, if the content at address 107C00 is equal to A200, it suggests that there is a high chance that the address 107C00 gets truncated to 7C00, resulting in us accessing the same memory location. However, it is still possible that the content at 107C00 coincidentally matches the value A200. To address this ambiguity, we perform a second test.

In this second test, we write a different random value, such as B200, to the address 7C00 and compare the content at 107C00 with B200. If the two values are still the same, it indicates that the A20 line is not enabled, and we proceed to the end of the test. We utilize the "JE" (Jump if Equal) instruction to determine the equality of the values.

Conversely, if the two values are not equal, it suggests that the A20 line is enabled, and we proceed to print the message "A20 line is on." Finally, we remember to change the ES register back to 0 by zeroing out the AX register and copying its value to the ES segment register.

Upon saving the file and building the project in the terminal, we can execute the A20 line test and determine its status.





Video Mode

Once we switch to long mode, we lose the ability to directly call BIOS functions. Therefore, in our project, we set up a video mode to display characters on the screen. Among the various video modes available, we choose the text mode, as it offers simplicity when it comes to printing characters.

To print a character on the screen, we only need to specify the ASCII code for the desired character and the attributes associated with it, such as the foreground and background colors. By writing these values to a specific memory address, the corresponding character will be displayed on the screen.

This approach allows us to interact with the user and provide visual feedback within the context of our project. Although we cannot utilize BIOS functions in long mode, implementing a text mode provides a straightforward and effective way to communicate information and instructions to the user.

To set up the text mode for printing characters, we use the interrupt 10h, with function code 0 in the AH register. This function code signifies our intention to set the video mode. In this case, we choose text mode by assigning the value 3 to the AL register. Therefore, we set the AX register to 3 to establish the desired text mode.

Once we have successfully set up the text mode, we can print messages on the screen without relying on BIOS services. To print characters in text mode, we need to understand the memory layout. The base address for text mode is b8000, and the screen size allows for 80 columns and 25 lines of text. Each character occupies 2 bytes of memory.

Starting from the first position on the screen, the corresponding memory address is b8000. The second position corresponds to b8002, and so on. Within each 2-byte

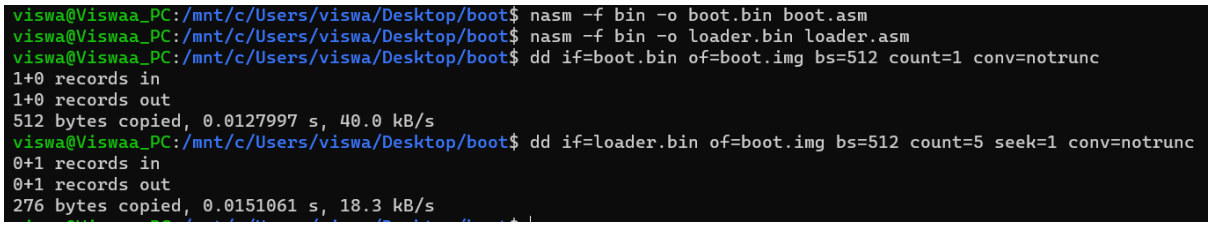
memory space, the first byte represents the ASCII code for the character, while the second byte represents the attribute of the character. The lower half of the attribute byte indicates the foreground color, and the higher half represents the background color. The color index determines the actual color displayed on the screen.

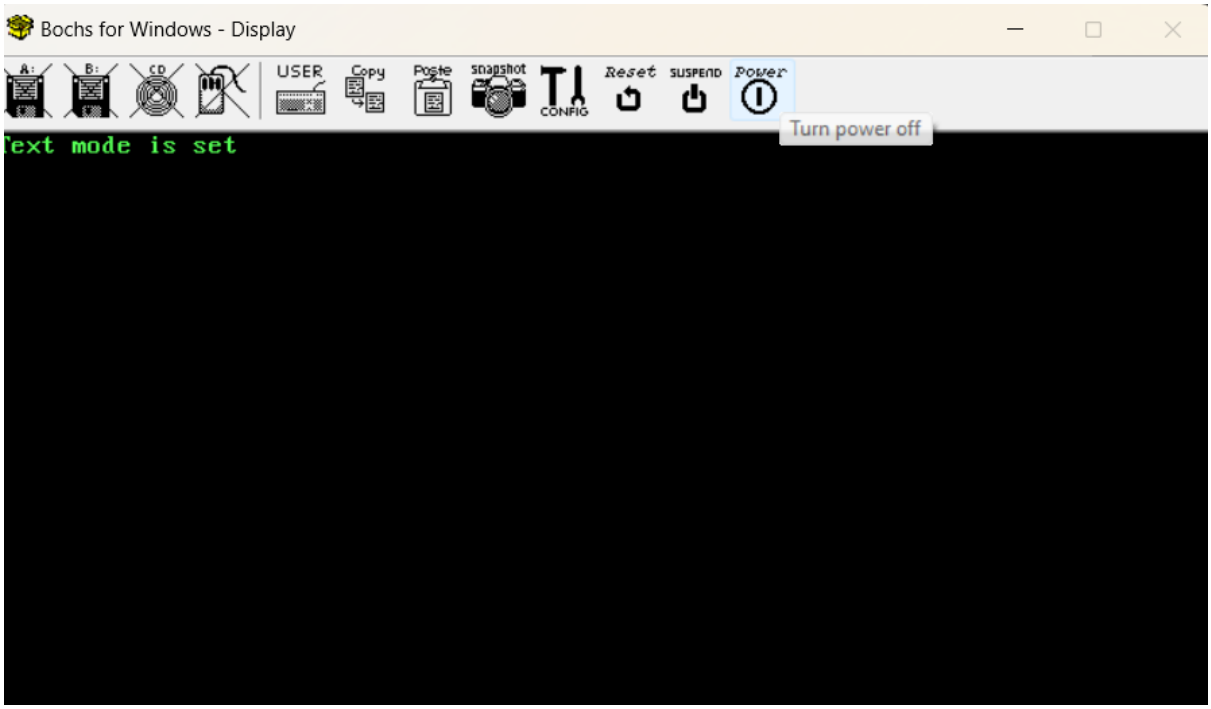
To print a message, such as "Text mode is set," we store the address of the message in the SI register and the text mode address b8000 in the DI register. Since the value b8000 is too large for a 16-bit DI register, we save b800 to the ES register and set DI to zero. By referencing the memory address b8000 using the ES:DI addressing mode, we effectively point to the same memory location.

We also save the number of characters in the CX register. Next, we print characters one at a time. We move the data from the memory location pointed to by SI, which represents the first character of the message, and copy it to the memory address pointed to by DI (b8000). In the next byte of memory, we specify the attribute of the character. In this case, we use color index a, which corresponds to a bright green color.

By adjusting the values of DI and SI registers, we move to the next character in memory. Since each character takes up two bytes and the character stored in the message only occupies one byte, we increment DI by 2 and SI by 1. This allows us to print characters sequentially. The loop instruction uses the CX register as a counter, ensuring that the block of code is executed the same number of times as the number of characters specified in CX.

In summary, by following this approach, we initialize the video mode, set up the memory layout, and print characters one by one with their corresponding attributes. This allows us to display messages and communicate information effectively on the screen in the chosen text mode.





Indeed, we were able to print a message on the screen without utilizing the BIOS service. This capability was achieved by setting up text mode and directly writing the character data and attributes to the memory address corresponding to the screen.

In real machines, when text mode is set, the video memory is mapped to a specific address range, typically starting at 0xB8000. Each character on the screen occupies 2 bytes of memory, where the first byte represents the ASCII code of the character, and the second byte represents the attribute (foreground and background color) of the character.

By storing the address of the character data in the SI register and the address of the video memory in the DI register, we can copy the character data and attributes from the SI address to the DI address, effectively displaying the desired text on the screen.

In this case, we printed the message "Text mode is set" by copying the corresponding character data and attribute to the video memory location at 0xB8000. We continued this process for each character in the message, incrementing the DI register by 2 bytes to move to the next character's position on the screen. By utilizing the CX register as a counter, we controlled the loop to iterate over all the characters in the message.

By leveraging this approach, we were able to directly manipulate the video memory and print the desired message on the screen without relying on BIOS services.

Protected Mode

Protected mode is a mode of operation in x86 processors that provides enhanced memory protection, multitasking capabilities, and extended addressing capabilities compared to real mode. Before transitioning to long mode, which allows for 64-bit operation, it is necessary to switch to protected mode first. In this report, we will discuss the key concepts related to protected mode and focus on the Global Descriptor Table (GDT) as a crucial component.

1. Real Mode and Protected Mode:
    - Real mode is the initial mode of operation when an x86 processor starts up. It provides a simple memory model, allowing direct access to all system memory but lacks memory protection and advanced features.
    - Protected mode, on the other hand, is a more sophisticated mode that offers memory protection, multitasking, and extended addressing capabilities.
2. Purpose of Protected Mode:
    - Protected mode is primarily used to enable memory protection, allowing the operating system to isolate processes and protect them from unauthorized access or corruption.
    - It also provides access to extended memory beyond the 1 MB limit of real mode, enabling larger and more complex applications to run.
3. Global Descriptor Table (GDT):
    - The Global Descriptor Table (GDT) is a data structure stored in memory that contains segment descriptors used by the CPU to access memory in protected mode.
    - Each entry in the GDT describes a segment or block of memory and occupies 8 bytes.
    - The first entry in the GDT should be empty to avoid conflicts with null pointers.
    - The number of entries in the GDT can exceed 8000, but for the purposes of this course, we only require a maximum of 5 entries.
4. Segment Registers and GDT Indexing:
    - In protected mode, the segment registers (e.g., DS, CS, ES) hold an index into the GDT, rather than the base address as in real mode.
    - When the processor accesses memory, the value in a segment register is treated as an index within the GDT to locate the appropriate segment

descriptor.

- This indexing mechanism is different from real mode, where the segment register value is shifted 4 bits and added to the offset to obtain the physical address.

5. Example: Conversion of DS and Effective Address:
   - Suppose the DS register holds the value 16, and we want to copy the value at the memory location 1000 to the EAX register.
   - To convert DS to a paragraph, we shift the value 4 bits to the left: 16 << 4 = 256.
   - The effective address is then calculated by adding the offset (1000) to the shifted segment register value: 256 + 1000 = 1256.
   - Consequently, the instruction `mov eax, ds:[1000]` copies the value from memory at the segment:offset address 1256 to the EAX register.

Protected mode is an essential mode of operation in x86 processors that provides memory protection and advanced features. The Global Descriptor Table (GDT) plays a crucial role in managing memory segments, and segment registers hold indices within the GDT in protected mode. The understanding of these concepts and the ability to convert segment register values to effective addresses are fundamental for implementing protected mode and transitioning to long mode.

Let us now focus on the Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), and privilege levels in x86 architecture. We will explore the structure and purpose of these tables and discuss their significance in protected mode.

1. Global Descriptor Table (GDT):
   - The Global Descriptor Table (GDT) is a memory structure created by the developer and used by the CPU to protect memory and manage segments in protected mode.
   - Each GDT entry is 8 bytes in size and describes a block of memory or segment.
   - The first entry in the GDT should be left empty to prevent conflicts with null pointers.

- The GDT can contain over 8000 entries, but for the purposes of this course, we only require a maximum of 5 entries.
- Segment descriptors in the GDT include information such as the base address of the segment, segment limit, and segment attributes.

2. Accessing Segments in Protected Mode:
    - In protected mode, the value in a segment register (e.g., DS, CS, ES) represents an index within the GDT rather than a base address, as in real mode.
    - To access memory, the processor uses the segment register value as an index to locate the corresponding segment descriptor in the GDT.
    - The segment descriptor contains information about the segment, including the base address.
    - The processor adds the offset to the base address to calculate the effective address for memory access.
3. Interrupt Descriptor Table (IDT):
    - An interrupt is a signal sent from a device to the CPU, prompting the CPU to stop the current task and handle the interrupt event.
    - Each interrupt is assigned a specific number, known as the Interrupt Request Number (IRQ).
    - The Interrupt Descriptor Table (IDT) is a structure created by the developer to store interrupt handlers or Interrupt Service Routines (ISRs).
    - The IDT can have a total of 256 entries, each corresponding to a specific interrupt.
    - Each IDT entry contains information about the segment where the ISR is located and the offset of the routine.
4. Privilege Levels (Rings):
    - The x86 architecture defines four privilege levels, often referred to as rings, ranging from ring 0 to ring 3.
    - Ring 0 is the most privileged level, where the kernel executes and has full access to instructions and hardware.

- Ring 3 is the least privileged level, where user applications execute and have limited access to instructions and hardware.
- In this course, we focus on utilizing two privilege levels: ring 0 for the kernel and ring 3 for user applications.

Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT) are essential components in protected mode. The GDT is used for managing memory segments and providing memory protection, while the IDT stores interrupt handlers for processing interrupt events. Understanding the structure and purpose of these tables, along with privilege levels, is crucial for implementing protected mode and developing operating systems.

Now we focuses on the process of enabling protected mode in x86 architecture. We will discuss the steps involved and examine the code used in a project to enable protected mode.

1. Disabling Interrupts:
    - Before switching to protected mode, it is necessary to disable interrupts to prevent interruptions during the mode switch.
    - The "clear interrupt flag" instruction is used to disable interrupts temporarily.
2. Loading the Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT):
    - The GDT and IDT structures, which hold segment and interrupt descriptors, respectively, are stored in memory.
    - The processor needs to be informed about the location of these structures by loading the Global Descriptor Table Register (GDTR) and Interrupt Descriptor Table Register (IDTR).
    - The "lgdt" instruction is used to load the GDTR with the address and size of the GDT structure.
3. Defining the GDT Entries:
    - The GDT structure contains multiple entries, each describing a segment or block of memory.
    - The first entry in the GDT is typically left empty.

- Each GDT entry is 8 bytes in size.
- In the example code, the first GDT entry is defined as empty, while subsequent entries are defined for specific segments.

4. Defining a Code Segment Descriptor:
    - The code segment descriptor is one of the segment descriptors within the GDT.
    - The code segment descriptor specifies attributes and characteristics of the code segment.
    - The descriptor is divided into several parts, including segment size, base address, and segment attributes.
    - The segment size is set to the maximum value (0xFFFF), indicating the entire code segment is usable.
    - The base address is set to 0, indicating the code segment starts from address 0.
    - The segment attributes are defined to indicate that it is a code segment and specify the type of code (non-conforming readable code segment in this example).

So enabling protected mode involves disabling interrupts, loading the GDT and IDT structures, and defining segment descriptors within the GDT. The GDT holds information about memory segments and provides memory protection, while the IDT stores interrupt handlers for processing interrupt events. By following the steps outlined and defining the necessary descriptors, the system can transition into protected mode.

In the next section, we will learn how to transfer execution from a lower privilege non-conforming code segment to a higher one in protected mode.

The Descriptor Privilege Level (DPL) indicates the privilege level of a segment. In order to run at ring 0 (highest privilege level) when switching to protected mode, we set the DPL to 0. When we load the descriptor into the cs register, the Current Privilege Level (CPL) will also become 0, indicating that we are at ring 0.

Moving on, the "P" (present) bit in the descriptor needs to be set to 1 when loading the descriptor. If this bit is not set, a CPU exception will be

generated.

The next bytes in the descriptor are a combination of segment size and attributes. The lower half represents the upper 4 bits of the segment size, which we set to the maximum size.

The "Available" bit can be used by system software and is ignored in this example.

The "DB" (default operand size) bit determines the default operand size. When this bit is set to 1, the default operand size is 32 bits. In protected mode, we set it to 1.

The "G" (granularity) bit is set to 1, indicating that the size field is scaled by 4 KB, allowing for a maximum segment size of 4 GB.

With all the necessary bits set, we assign the value "CF" to this byte.

The last byte represents the upper 8 bits of the base address, and we set it to 0.

This code segment descriptor will be used in protected mode. Similarly, a data segment descriptor is needed for accessing data in memory. The structure of the data segment descriptor is similar to that of the code segment descriptor, so we can simply copy and paste the code segment descriptor and make the necessary changes.

We name the data segment descriptor "data32", and the base address and size remain the same as the code segment. The only change is in the "type" field, which we set to 0010 in binary, indicating a readable and writable data segment.

It is important to set the writeable bit to 1 in order to write data into memory locations referenced by this segment descriptor. Failure to do so will result in an exception when attempting to write data.

With the code and data segment descriptors defined, we can calculate the length of the descriptor table. We define a constant called "gdt32_length" and assign the length of the table to the first two bytes, followed by the address of the GDT in the next four bytes.

Next, we load the GDT using the "lgdt" instruction and provide the address of the GDT pointer. It is worth noting that the default operand size of the "lgdt" instruction is 16 bits in 16-bit mode. However, in this case, we define the address to be 32 bits and assign the address of the GDT to the lower 24 bits.

Finally, we proceed to load the Interrupt Descriptor Table (IDT).

We also have an Interrupt Descriptor Table (IDT) register that needs to be loaded with the address and size of the IDT. We use the "load idt" instruction for this purpose. However, since we don't want to deal with interrupts until we jump to long mode, we load the register with an invalid address and size of zero.

To accomplish this, we define a variable called "idt32_pointer" and assign zero to it. We then load the IDT pointer using this variable.

It is important to note that there is one type of interrupt called a non-maskable interrupt, which is not disabled by the "cli" (clear interrupt flag) instruction. When a non-maskable interrupt occurs, the processor will still attempt to find the IDT in memory. Since we have provided an invalid address and size for the IDT, a CPU exception will be generated, resulting in a system reset. This is intentional because non-maskable interrupts typically indicate non-recoverable hardware errors such as RAM errors, and in such cases, it is best to reset the system.

Once we have loaded the GDT and IDT, we proceed to enable protected mode by setting the protected mode enable bit in the CR0 (Control Register 0) register. We accomplish this by copying the content of CR0 to the EAX register, setting bit 0 to 1 using the "or" instruction, and writing the modified value back to CR0. This enables protected mode.

The last step is to load the CS (Code Segment) register with the new code segment descriptor we defined in the GDT table. Loading the code segment descriptor into the CS register is different from loading other segment registers. We cannot use the "mov" instruction to load the CS register. Instead, we use a jump instruction to accomplish this. In this example, the code segment descriptor is the second entry in the GDT, located 8 bytes from the beginning of the GDT. So, we use a selector index of 8, TI of 0 (indicating the use of GDT), and an RPL (Requested Privilege Level) of 0. These values ensure that the RPL matches the DPL (Descriptor Privilege Level) of the code segment, allowing the privilege checks to pass. We also specify the offset to jump to, which is the protected mode entry label that we will define.

Before defining the protected mode entry label, we use the "bits" directive to indicate that the following code is running in 32-bit mode.

Next, we define the protected mode entry label and proceed to initialize other segment registers such as DS, ES, and SS. We load them with the data segment descriptors, with the data segment descriptor being the third entry in the GDT (index 16 or 10 in hexadecimal). For initializing these segment registers, we can use the "mov" instruction.

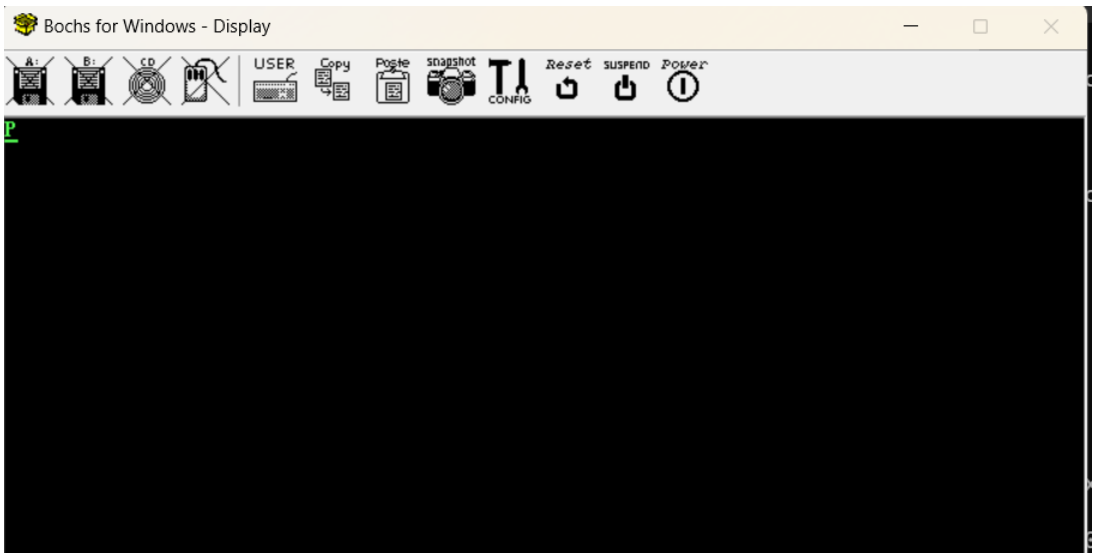
We then set the stack pointer ESP to 0x7C00 and print the character 'P' on the screen to indicate that we have entered protected mode. In text mode, the base address is 0xB8000, and each character takes up 2 bytes of space. The first byte represents the ASCII code 'P', and the second byte represents the attribute of the character.

Finally, we define a label called "pend" and halt the processor using an infinite loop, just as we did before.

Since we were only printing the character 'P', the message in this section is not used.

This block of code is used before we jump into protected mode, so it should be placed at the end of the code in 16-bit mode.

```
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o boot.bin boot.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o loader.bin loader.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=boot.bin of=boot.img bs=512 count=1 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00529075 s, 96.8 kB/s
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=loader.bin of=boot.img bs=512 count=5 seek=1 conv=notrunc
0+1 records in
0+1 records out
323 bytes copied, 0.0082258 s, 39.3 kB/s
```



Long Mode

Now that we are in protected mode, we will proceed to prepare and switch to long mode. Long mode consists of two sub-modes: 64-bit mode and compatibility mode, with compatibility mode being unused in our system.

When we switch to long mode, we are actually running in 64-bit mode. Therefore, in this course, when we refer to long mode, we are specifically talking about 64-bit mode. Both our kernel and application programs will run in this mode. Once we jump to the kernel entry, we will remain in long mode and never leave it until we shut down the system.

It's important to note that protected mode is only used to prepare for long mode. Immediately after jumping to protected mode, we will begin the process of preparing and switching to long mode. This process is slightly more complex than preparing for protected mode.

Preparing for long mode involves several steps, such as loading the GDT and IDT, enabling paging, setting specific registers, and loading the new code segment descriptor in the CS register using a jump instruction.

In 64-bit mode, the GDT is still used, but the fields in GDT entries are slightly different. In terms of segment registers, the contents of DS, ES, and SS registers are ignored, and their memory segments are treated as if the base address is 0 and the segment size is the maximum size the processor can address. However, there is a special case when transitioning from ring 0 to ring 3 to run user applications where a valid data segment descriptor is needed to load into the SS register.

It's important to note that although the contents of DS, ES, and SS registers are ignored, the processor still performs checks to validate the descriptors when loading them into these registers.

In 64-bit mode, the IDT is still used, and each entry takes up 16 bytes compared to the 8-byte entries in protected mode. The process of interrupt handling remains the same as in protected mode.

The privilege levels (ring levels) in 64-bit mode follow the same principles as discussed previously, with the kernel and user programs running in ring 0 and ring 3, respectively, while rings 1 and 2 are unused.

Loading segment descriptors into segment registers in 64-bit mode is less common, as the code segment descriptors are loaded when jumping to 64-bit mode, and the data segment descriptor in the SS register is loaded when transitioning to ring 3 for the first time.

The descriptor formats for code and data segment descriptors in 64-bit mode have specific bits that need to be set. The attributes field has limited bits that can be set, with the dark grey fields being ignored. The C (conforming) bit is set to 1 for non-conforming code segments. The DPL (privilege level) is set to 0. The P (present) bit should be 1 to avoid CPU exceptions. The L (long) bit is set to 1 to indicate running in 64-bit mode, and the D bit is set to 0. For data segment descriptors, the W (writable) bit is set to 1, and the P and DPL fields have the same meaning as in code segment descriptors.

In terms of memory addressing, there are virtual addresses and physical addresses. Virtual addresses are what we work with in our code and they need to be mapped to physical addresses using the Memory Management Unit (MMU). Physical addresses are the addresses put on the memory bus to access data in memory locations.

In 64-bit mode, we have a 64-bit virtual address space, but not all virtual addresses are available to us. The concept of a canonical address is introduced, which refers to an address where the bits from 63 through to the most-significant implemented bit are either all ones or all zeros.

For example, if we have a 48-bit address, with bits 0 to 47, the most-significant implemented bit is bit 47. In this case, we can only set the upper 16 bits (bits 48 to 63) to all ones. Any other value in the upper 16 bits would be considered an invalid or non-canonical address. Similarly, if bit 47 is set to 0, we can only set the upper 16 bits to all zeros.

Using a non-canonical address typically results in a CPU exception. It's worth noting that recent processors have added support for 57-bit virtual addresses, which can address a memory space of up to 128 petabytes. However, for this course, we assume that the processors support only 48-bit virtual addresses to ensure compatibility with older machines.

Now let's consider the memory map setup for our system in 64-bit mode. We have the user space and kernel space available in the address space. The addresses marked in red in the memory map represent the address range where all the addresses are non-canonical and cannot be used. You can verify this by examining bit 47 and the upper 16 bits of these addresses, which don't conform to the canonical address format. The kernel will be placed in the higher part of the memory space.

Starting from protected mode, we remove the code for printing characters. The first step in enabling long mode is to set up paging. Enabling long mode

requires us to set up and enable paging, which will be discussed in more detail later in the course. For now, we will write the code as it is.

Code:

boot.asm

```
[BITS 16]
[ORG 0x7c00]

start:
    xor ax,ax
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov sp,0x7c00

TestDiskExtension:
    mov [DriveId],dl
    mov ah,0x41
    mov bx,0x55aa
    int 0x13
    jc NotSupport
    cmp bx,0xaa55
    jne NotSupport

LoadLoader:
    mov si,ReadPacket
    mov word[si],0x10
    mov word[si+2],5
    mov word[si+4],0x7e00
    mov word[si+6],0
    mov dword[si+8],1
    mov dword[si+0xc],0
    mov dl,[DriveId]
    mov ah,0x42
    int 0x13
    jc  ReadError

    mov dl,[DriveId]
    jmp 0x7e00

ReadError:
NotSupport:
    mov ah,0x13
    mov al,1
    mov bx,0xa
    xor dx,dx
    mov bp,Message
    mov cx,MessageLen
    int 0x10

End:
    hlt
    jmp End
```

```
DriveId:    db 0
Message:    db "We have an error in boot process"
MessageLen: equ $-Message
ReadPacket: times 16 db 0

times (0x1be-($-$$)) db 0

    db 80h
    db 0,2,0
    db 0f0h
    db 0ffh,0ffh,0ffh
    dd 1
    dd (20*16*63-1)

    times (16*3) db 0

    db 0x55
    db 0xaa
```

loader.asm

```
[BITS 16]
[ORG 0x7e00]

start:
    mov [DriveId],dl

    mov eax,0x80000000
    cpuid
    cmp eax,0x80000001
    jb NotSupport

    mov eax,0x80000001
    cpuid
    test edx,(1<<29)
    jz NotSupport
    test edx,(1<<26)
    jz NotSupport

LoadKernel:
    mov si,ReadPacket
    mov word[si],0x10
    mov word[si+2],100
    mov word[si+4],0
    mov word[si+6],0x1000
    mov dword[si+8],6
    mov dword[si+0xc],0
    mov dl,[DriveId]
    mov ah,0x42
    int 0x13
    jc  ReadError

GetMemInfoStart:
    mov eax,0xe820
    mov edx,0x534d4150
```

```
    mov ecx,20
    mov edi,0x9000
    xor ebx,ebx
    int 0x15
    jc NotSupport

GetMemInfo:
    add edi,20
    mov eax,0xe820
    mov edx,0x534d4150
    mov ecx,20
    int 0x15
    jc GetMemDone

    test ebx,ebx
    jnz GetMemInfo

GetMemDone:
TestA20:
    mov ax,0xffff
    mov es,ax
    mov word[ds:0x7c00],0xa200
    cmp word[es:0x7c10],0xa200
    jne SetA20LineDone
    mov word[0x7c00],0xb200
    cmp word[es:0x7c10],0xb200
    je End

SetA20LineDone:
    xor ax,ax
    mov es,ax

SetVideoMode:
    mov ax,3
    int 0x10

    cli
    lgdt [Gdt32Ptr]
    lidt [Idt32Ptr]

    mov eax,cr0
    or eax,1
    mov cr0,eax

    jmp 8:PMEntry

ReadError:
NotSupport:
End:
    hlt
    jmp End


[BITS 32]
PMEntry:
    mov ax,0x10
    mov ds,ax
    mov es,ax
```

```
    mov ss,ax
    mov esp,0x7c00

    cld
    mov edi,0x70000
    xor eax,eax
    mov ecx,0x10000/4
    rep stosd

    mov dword[0x70000],0x71007
    mov dword[0x71000],10000111b

    lgdt [Gdt64Ptr]

    mov eax,cr4
    or eax,(1<<5)
    mov cr4,eax

    mov eax,0x70000
    mov cr3,eax

    mov ecx,0xc0000080
    rdmsr
    or eax,(1<<8)
    wrmsr

    mov eax,cr0
    or eax,(1<<31)
    mov cr0,eax

    jmp 8:LMEntry

PEnd:
    hlt
    jmp PEnd

[BITS 64]
LMEntry:
    mov rsp,0x7c00

    mov byte[0xb8000],'L'
    mov byte[0xb8001],0xa

LEnd:
    hlt
    jmp LEnd



DriveId:    db 0
ReadPacket: times 16 db 0

Gdt32:
    dq 0
Code32:
    dw 0xffff
    dw 0
    db 0
```

```
    db 0x9a
    db 0xcf
    db 0
Data32:
    dw 0xffff
    dw 0
    db 0
    db 0x92
    db 0xcf
    db 0

Gdt32Len: equ $-Gdt32

Gdt32Ptr: dw Gdt32Len-1
          dd Gdt32

Idt32Ptr: dw 0
          dd 0


Gdt64:
    dq 0
    dq 0x0020980000000000

Gdt64Len: equ $-Gdt64


Gdt64Ptr: dw Gdt64Len-1
          dd Gdt64
```

In this code block, we perform the necessary steps to enable 64-bit mode (long mode) and load the code segment descriptor. Let's go through the code step by step:

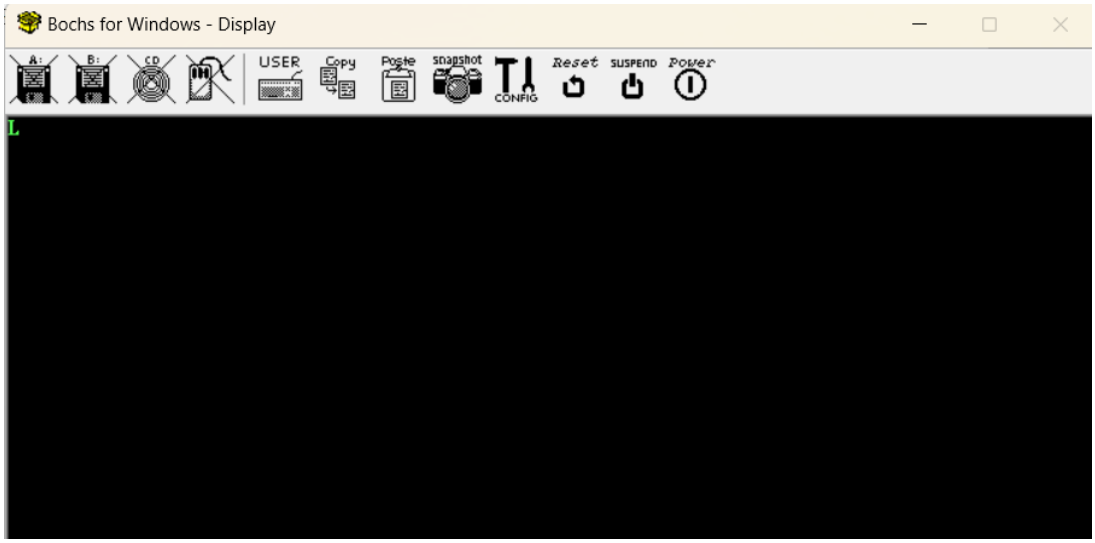
1. We set up paging, which involves finding a free memory area and initializing the paging structure used to translate virtual addresses to physical addresses.
2. We load the Global Descriptor Table (GDT) that will be used in 64-bit mode. We define a GDT pointer called "gdt64 pointer" that specifies the length and address of the GDT.
3. We define the GDT entries, starting with an empty entry (assigned as 0) and then the code segment entry. Most of the fields in the code segment entry are ignored, and we set the necessary attributes such as the privilege level (DPL) and the long bit (indicating 64-bit mode).
4. We load the GDT pointer using the "lgdt" instruction.
5. We enable 64-bit mode by setting the necessary bits in some registers. We set bit 5 (PAE bit) in the Control Register 4 (CR4) register to 1, and

we set the address of the page structure we set up earlier in the Page Directory Base Register (CR3) register.

6. We enable long mode by setting the long mode enable bit (bit 8) in the Extended Feature Enable Register (MSR). We use the "rdmsr" and "wrmsr" instructions to read and write the MSR.
7. Finally, we enable paging by setting bit 31 in the Control Register 0 (CR0) register.
8. We load the new code segment descriptor by jumping to the long mode entry, using the segment selector 8 and the offset of the long mode entry label. We define the label "long mode entry" and specify the directive "bits" to indicate that we are now running in 64-bit mode.
9. At the long mode entry label, we initialize the stack pointer (RSP) to 7C00 and print the character 'L' on the screen to indicate successful entry into long mode.
10. After printing the character, we end the system by jumping to the infinite loop label "Lend".

That's the overview of the code flow in this block, which enables long mode and loads the necessary code segment descriptor to run in 64-bit mode.

```
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o boot.bin boot.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ nasm -f bin -o loader.bin loader.asm
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=boot.bin of=boot.img bs=512 count=1 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00505623 s, 101 kB/s
viswa@Viswaa_PC:/mnt/c/Users/viswa/Desktop/boot$ dd if=loader.bin of=boot.img bs=512 count=5 seek=1 conv=notrunc
0+1 records in
0+1 records out
446 bytes copied, 0.00491782 s, 90.7 kB/s
```



Conclusion

In conclusion, the code block provided demonstrates the steps required to enable and enter 64-bit mode (long mode) in the context of operating system development. The key points covered are:

1. Setting up paging: Initializing the paging structure and enabling paging, which allows for memory management and protection in 64-bit mode.
2. Loading the Global Descriptor Table (GDT): Defining the GDT entries, particularly the code segment descriptor, which specifies the attributes and privileges for executing code in 64-bit mode.
3. Enabling long mode: Setting the necessary bits in registers, such as enabling Physical Address Extension (PAE) and activating long mode through the Extended Feature Enable Register (MSR).
4. Loading the code segment descriptor: Jumping to the long mode entry point and loading the new code segment descriptor by specifying the segment selector and offset.
5. Initialization and termination: Setting up the stack pointer and printing a character to confirm successful entry into long mode. Finally, ending the system by entering an infinite loop.

By following these steps, the system transitions from protected mode to 64-bit mode, allowing for the execution of 64-bit code and utilizing the larger address space and extended capabilities provided by the processor.