DOCUMENTATION

Keyboard Drivers:

1. What is a Keyboard Driver?
   Keyboard is the primary input device for a computer. Working with a keyboard is a non avoidable factor in os development.

2. Why is it needed?
   To implement the keyboard,we need to ask every time if there is any input or not. But this isn't an efficient way, as the keyboard is much slower than the processor. Asking it everytime will affect the processing speed badly. And it also will put a lot of traffic in the system bus.

3. How is it implemented?
   The solution implemented by the x86 chip manufacturer uses interrupts.

   Whenever a key is being pressed, The cpu will call a function that we pre define to it. The processor won't do the keyboard logic, but it will let us execute some code whenever a key is being pressed.

   In simple words, the code to handle the keyboard input passes its address to the interrupt descriptor table and tells the processor to load it. After loading all the parameters and the location of our keyboard handling code, It will call that code whenever a key is being pressed.

   When we get this interrupt, We could try reading from the keyboard which gives us the key being pressed.

   Note: What keyboard gives as the value for pressed key is not ascii. It is called scan codes. The OS converts it to ascii in the kernel.

   The PIC Chip
   PIC or **Programmable Interrupt Controller** is a chip in the computer whose main job is to generate interrupts. When a key in keyboard is pressed, The Chip inside keyboard tells to the pic chip inside our computer to generate a #1 Interrupt. The pic chip will then decide the time to notify the cpu about the interrupt. When the cpu gets the message which says a key is being pressed, It executes a set of code to handle the interrupt.

4. Code Explanation.
   In the `main.c` file, in the **`start()`** function, the kernel calls the `initIDT()` function. The job of this function is to set the Interrupt Descriptor Table so that when a

key is pressed, It will call the `isr1_Handler()` function.

1. The struct `IDT_ENTRY` could be used to define the `idt`. There are a total of **256** possible idt entries. So we created 256 of them with `struct IDT_ENTRY idt[256]`

```
struct IDT_ENTRY{
unsigned short base_Lower;
unsigned short selector;
unsigned char zero;
unsigned char flags;
unsigned short base_Higher;
};
struct IDT_ENTRY idt[256];
unsigned int base;
```

2. The next command extern unsigned int isr1; says to the compiler to implement compilation so that it knows that there will be a section named isr1 in an outside source file. We could find this section in the IDT.asm file.

   ```
   extern unsigned int isr1;
   ```

3. In the `start()` function in `main.c`, we copied the address of the `isr1` section to the variable `base` with `base = (unsigned int)&isr1;` (We used the & Symbol to get the address).

4. In the `initIDT()` function, we mapped the #1 interrupt(Keyboard interrupt(IRQ 1)) with:

   ```
   idt[1].base_Lower = (base & 0xFFFF);
   idt[1].base_Higher = (base >> 16) & 0xFFFF;
   idt[1].selector = 0x08;
   idt[1].zero = 0;
   idt[1].flags = 0x8e;
   ```

   Here we set the location of the `isr1` section in the first two lines using the value in `base` variable.
   The next three variables set the arguments necessary for the idt entry.

5. Then we prepared the pic chip to handle interrupts and finally, we called the `loadIdt()` function. At the very top of the code, we said to the compiler that the `loadIdt()` function will be outside the c source file with `extern void`

```
loadIdt();
```

6. When the loadIdt() code is executed, it jumps to that section we defined in the IDT.asm file.

   ```
   _loadIdt:

         lidt[idtDesc]
         sti
         Ret
   ```

   `lidt` is a x86 command used to make the processor load the Interrupt Descriptor Table. Here, it loads the idt entry Which the `idtDesc` section points to.
   The command `sti` is used to enable the interrupts. From now onwards , we will get every interrupt.

7. From now onwards, the `_isr1` section in IDT.asm file will be called when ever a keyboard key is being pressed.

8. The `_isr1` section calls the `_isr1_Handler` function we defined in the `main.c` file

   Whenever a key is pressed, the function `isr1_Handler()` will be called. We could implement whatever to do in that function.

9. According to our implementation we first obtained the scan code of key being pressed with `inportb(0x60)` and passed it to the handleKeypress() function with `handleKeypress(inportb(0x60));`

10. In the `handleKeypress()` function, we have an array of values named `Scancode[].`
    We could take the scan code as an index to point to the Scancode[] array

    The arrangement of values in `Scancode[]` gives us the ascii representation of the scan code.

11. Finally we called a function to print the character being pressed.

    ```
    THE inportb AND outportb FUNCTIONS ARE FUNCTIONS THAT HELP
    US
    COMMUNICATE WITH EXTERNAL DEVICES. in COMMAND ACCEPTS INPUT
    FROM EXTERNAL DEVICES AND out COMMAND OUTPUTS COMMANDS AND
    DATA TO EXTERNAL DEVICES.
    ```

Compile and run the code by executing the `compile.bat` file and try pressing any character on keyboard, The program simply prints it to the screen.

Audio:

1. How is Audio Implemented?

Generating audio is an interesting part when doing a project. We actually give some integer value(Representing the frequency) which will turn to an audible effect. We directly communicate with the sound card using our command and the sound card works on further stuff.

2. Implementing a small 'beep' sound:

1. Add the following argument when calling qemu in `compile.bat`:

```
-soundhw pcspk
```

```
compile.bat
```

2. main.c

```
unsigned char inportb(unsigned short _port){
unsigned char rv;
__asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
return rv;
}

void outportb(unsigned short _port, unsigned char _data) {
__asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
}

void play(unsigned int fr) {
unsigned int Div;
unsigned char tmp;

Div = 1193180 / fr;
outportb(0x43, 0xb6);
outportb(0x42, (unsigned char) (Div) );
outportb(0x42, (unsigned char) (Div >> 8));

tmp = inportb(0x61);
if (tmp != (tmp | 3)) {
outportb(0x61, tmp | 3);
}
}

void stop() {
unsigned char fr = inportb(0x61) & 0xFC;
outportb(0x61, fr);
}

void scream(int fr) {
play(fr);
int Delay = 10000;
while(Delay--);
stop();
}

int start(){
scream(2020);
}
```

Here , we called the `scream` function with `scream(2020)`. We could try giving different values as an argument to the scream function to generate different sounds.

Video Player

1. What is a video player?

Videos are actually moving pictures. When we see multiple pictures during a small amount of time, Our brain takes it as moving or gives us a sense of The so called "Video".

2. How is it implemented?

Changing the data contained in the video memory continuously gives us this effect

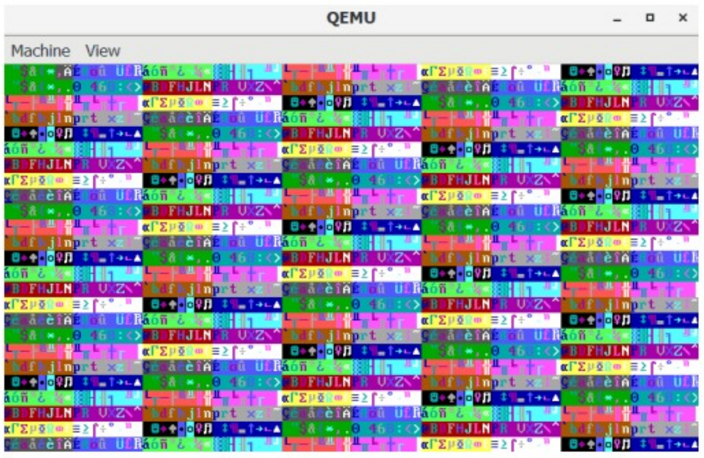
But the processor is much faster than our video card and the monitor. So it won't be able to display every frame the processor generates very fastly. So the solution is to minimize the frame updating procedure depending on the refresh rate of the monitor.

But the computer must also be fast enough to play about `60` frames a second. Human eyes see upto `60` frames per second. So there's no point in generating video with frames more than `60` per second.

3. Practical implementation:

```
int start(){
    char* TM_START = (char*) 0xb8000;
    int i;
    char obj = 0;
    while(1){
            i = 0;
            while(i < (2 * 80 * 25)){
                    *(TM_START + i) = obj;
                    i++;
                    obj++;
            }
    }
}
```

4. One Frame Of Output During Video Playing:



The program is in an infinite loop with `while(1)`.

1) What is GDT

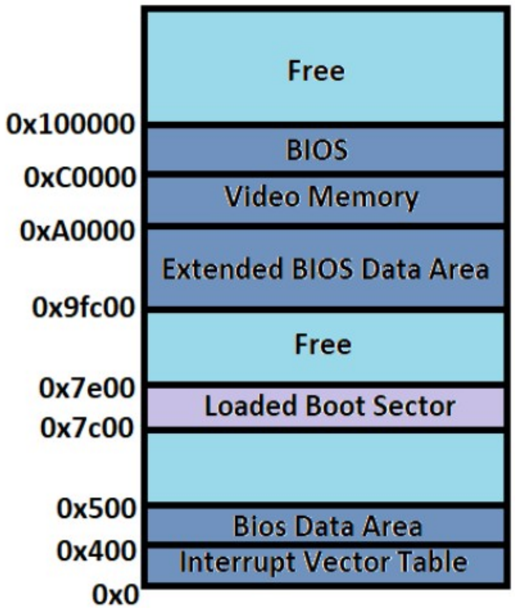
Global Descriptor Table or GDT is a table which contains information about memory areas and who can access those. This is very essential as an absence of this feature could allow user mode programs to do things like editing the kernel routines which results in complete system takedown.

2) Why is GDT required

Switching to 32 bit mode allows us protect some memory locations from other user mode programs trying to access those location. If a user mode program could access these special locations where the Kernel of the os resides, Those programs could take control of the CPU so our OS would not have any control.

3)Implementing the GDT

We can define which part of the memory area to be protected by defining it in a table named GDT or Global Descriptor Table. It is necessary to Set The GDT Before switching to 32 bit protected mode. After defining and loading the GDT, We could switch the cpu to 32 bit mode.



Boot sector is present, is at 0x7c00.

This means that Bios will load our Boot sector to 0x7c00 in memory.

Before switching the processor to 32 bit mode, We need to define our Global Descriptor Table(GDT). GDT is a special data Structure which the processor directly validates.

4)**Code Explanation**

4.1)[org 0x7c00] Says to the assembler to make adjustments to the program so that it could calculate 0x7c00 as the address which it will be loaded to.

4.2)[bits 16] says to produce 16 bit opcodes for instructions given below it.

4.3) cli turns the interrupt services off.

4.4)We loaded the GDT we defined at the bottom of the code with: lgdt

[GDT_DESC]

4.5) **GDT_BEGIN and GDT_NULL_DESC gives us the address to that location which we could later refer. dd defines two null pointer(0x0)**.

```
GDT_BEGIN:

GDT_NULL_DESC:

dd 0x0

dd 0x0
```

4.5)**Defining data segment,the code tells the CPU about the code segment and we defined base,limit and some flags**.

```
GDT_DATA_SEG:

dw 0xffff ;Limit

dw 0x0 ;Base

db 0x0 ;Base

db 10010010b ;Flags

db 11001111b ;Flags

db 0x0 ;Base
```

4.6) **dw GDT_END - GDT_BEGIN - 1 stores the size of our GDT Entry**

dd GDT_BEGIN Stores the beginning address of our gdt entry.

```
GDT_DESC:

dw GDT_END - GDT_BEGIN - 1

dd GDT_BEGIN
```

4.7)**The equ command assigns the Result of GDT_CODE_SEG - GDT_BEGIN(Substraction operation) to CODE_SEG.**

```
CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN

DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
```

4.8)**This code sets the cr0 bit. This will switch the processor to 32 bit**

protected mode.

```
mov eax , cr0

or eax , 0x1

mov cr0 , eax
```

4.9)**We implement a farm jump and implemented it with jmp CODE_SEG:INIT_PM, which clears the contents in cache memory and jump to the section named INIT_PM.section:**

```
[bits 32]

INIT_PM:

mov ax , DATA_SEG

mov ds , ax

mov ss , ax

mov es , ax

mov fs , ax

mov gs , ax

mov ebp , 0x90000
mov esp , ebp
```

4.10) [bits 32] tells to the assembler to generate 32 bit opcodes for the instructions following it. We used the mov command to move the address of data segment to all of the Segment Registers. We set the esp and ebp registers. The C Programs we later make will push values to the stack.

4.11)jmp $ is used to stop processor from executing further code. This is technically a jump to that jmp instruction itself.

5) References

5.1)https://en.wikipedia.org/wiki/Global_Descriptor_Table

5.2)https://wiki.osdev.org/Global_Descriptor_Table

5.3https://wiki.osdev.org/GDT_Tutorial

1)What is IDT

Interrupt Descriptor Table or IDT is a mechanism implemented in x86 cpu which allows an operating system to capture events and do necessary operations. The events include Keyboard Input , Mouse Input etc… and also includes software defined interrupts which in x86 cpu occurs during things like when executing the int command.

2)**How it works**

int is a command in assembly known as Interrupt. This command is used to pause the current execution and execute another code which is defined in Interrupt Descriptor Table.

For the system to know which interrupt service routine to call when a certain interrupt occurs, offsets to the ISRs are stored in the Interrupt Descriptor Table when you're in Protected mode.

3)**Code Explanation**

3.1) initIDT() function sets the IDT so that it will call the isr1_Handler() function when a key is pressed.

```
struct IDT_ENTRY{

unsigned short base_Lower;

unsigned short selector;

unsigned char zero;

unsigned char flags;

unsigned short base_Higher;

};

struct IDT_ENTRY idt[256];

extern unsigned int isr1;

unsigned int base;
```

3.2) IDT_ENTRY define the idt. we created 256 idt entries with

```
struct IDT_ENTRY idt[256];
```

3.3) extern unsigned int isr1; tells that there will be a section named isr1 in an outside source file.This section is in the IDT.asm file.

```
extern _idt

extern _isr1_Handler

global _isr1

global _loadIdt
```

```
idtDesc:

dw 2048

dd _idt

_isr1:

pushad

call _isr1_Handler

popad

iretd

_loadIdt:

lidt[idtDesc]

sti

ret
```

3.4) We copied the address of the isr1 section to the variable base with base = (unsigned int)&isr1

3.4) we mapped the #1 interrupt(Keyboard

interrupt(IRQ 1)) with:

```
idt[1].base_Lower = (base & 0xFFFF);idt[1].base_Higher = (base >> 16) & 0xFFFF;

idt[1].selector = 0x08;

idt[1].zero = 0;

idt[1].flags = 0x8e;
```

3.5) We set the location of isr1 section using the value in base variable,and the next 3 variables sets the arguments necessary for the idt entry.

3.6) The loadIdt() function wil be outside the c source file with extern void

When the loadIdt() code is executed, it jumps to that section we defined in the IDT.asm file.

```
_loadIdt:

lidt[idtDesc]

sti

ret
```

lidt is a x86 command used to make the processor load the IDT which the idtDesc section points to.

The command sti is used to enable the interrupts.

3.7) The _isr1 section calls the _isr1_Handler function we defined in the main.c file.When ever a key is pressed, the fuction isr1_Handler() will be called.

3.8) We first obtained the scan code of key being pressed with inportb(0x60) and passed it to the handleKeypress()function with handleKeypress(inportb(0x60));

3.9) In the handleKeypress() function, we have an array of values named Scancode[].

We take the scan code as an index to point to the Scancode[] array.

The arrangement of values in Scancode[] gives ascii representation of the scan code.

3.10) Finally we called a function to print the character being pressed.

3.11)Then we compile the code by executing the compile.bat file and pressing any key on the keyboard(The inportb and outportb functions are functions that help us communicate with external devices. In command accepts input from external devices and out command outputs commands and data to external devices.)

4)**References**

4.1)https://wiki.osdev.org/Interrupts

4.2)https://wiki.osdev.org/Interrupt_Descriptor_Table

4.3)https://wiki.osdev.org/Interrupt_Service_Routines

1)What is HDD

Hard Disk Drive or HDD is a mechanical hard disk with a rotating platter and a moving head. Data is stored into it in magnetic form. The Platter(Where data is stored) , rotates during read or write operations and the Head(The part of hard disk which do the read/write operation) performs the desired operation on the platter.

2)**what is SSD**

SSD or Solid State Drive is a non mechanical type of hard disk where the data is stored typically using flash memory technology. As it is non mechanical , SSD's are much faster. SSD's will give us a faster boot time , load time and even increase the process execution speed. This is because the paging operation will be much faster. Paging is a concept used to utilize main memory to its fullest by putting "less often used" data temporarily to hard Hard disk and later copy it back to memory when needed.

3)How the HDD is organized to store data.

The data is stored in magnetic form in platters. Typically hard disks have one to four platters stacked together. All of this platters will have separate Heads to read/write data.

All of this platters are further divided into tracks and sectors. There are number of tracks in a hard disk. All of this tracks will have a set of sectors in it. Usually the size of one sector is 512 bytes.

4)**How Hard Disk is Divided**

The data is stored in magnetic form in platters. Typically hard disks have one to four platters stacked together. All of this platters will have separate Heads to read/write data.

All of this platters are further divided into tracks and sectors. There are number of tracks in a hard disk. All of this tracks will have a set of sectors in it. Usually the size of one sector is 512 bytes.

Hard disks have number of platters and each platters are further divided into tracks. A cylinder is formed by joining the same tracks in all of the platters. For eg: Track 1 of Platter 1 + Track 1 of Platter 2 + Track 1 of Platter 3 ……… Forms one cylinder.

5)How it's implemented

5.1)Here, we implemented a hard disk driver for the ATA technology. ATA or Advanced Technology Attachment allows hard disks and CD-ROMs to be internally connected to the motherboard and perform input/output functions.

5.2)We use the LBA method for this since this is the easiest way to read/write to hard disk , all we need to do is pass the Block address of sector. Passing 0 will give us access to the firstsector(Boot sector).

5.3)The two new commands we added to the strEval function are GET and PUT. When enter the PUT command, it first copies the number 0 to blockAddr and then proceeds to initialize every cell in At character array with 'J' and finally adds a null character. Then it calls the put function which calls another function named write.

5.4)The write function is defined in ata.asm file.

```
mov eax , [_blockAddr]

mov cl , 1

mov edi , _At
```

5.5)The program copies the value we copied to the blockAddr variable in strEval function to the eax register. This value represents which sector to write to hard disk.

5.6)Then we copied the value 1 to cl register , this represents the number of sectors to write. As we give 1 to that register, it writes 1 sector(512 bytes).

5.7)We copied the address of the array At we defined in extra.h file so that the processor will write the data in that array to hard disk. The rest of the code communicates with the hard disk to write to it, and after the operation ends, it returns to our code in C.

5.8)The same process happens when we give the GET command. But instead of the writing process , it does the reading process. In the code , we first copied the value 0 to blockAddr variable, Then we called the get function which calls the read function in ata.asm file. That function reads the 0‘th sector to the At array and returns to the code in C. and finally prints the content in that array.Now , when running it, try giving the GET command first. We will get some random characters as output.

5.9)Giving the PUT command and later calling the GET command will print a lot a J to the screen because , when we give the PUT command , it copies the string of J’s to At array and writes it to the hard disk. After that , when we call the GET command , it reads from the hard disk and outputs the string of J’s we previously copied to it. We done all of these operations to the 0‘th sector. We could change the value in blockAddr variable to read/write to other sectors.

6)**Code**

```
else if(strcmp(CMD , cmd8)){

blockAddr = 0;

int i = 0;

while(i < 511){

At[i] = 'J'; // Fill with J
i++;
}
At[i] = 0; // Null character
put(); // Writes to Hard disk
i = 0;
while(i < 511){
At[i] = 0; // Clears the content
i++;
}
}
else if(strcmp(CMD , cmd9)){
blockAddr = 0;
get();
printString(At);
```

6)**References**

6.1)https://wiki.osdev.org/Category:ATA

6.2)https://wiki.osdev.org/ATA_PIO_Mode

PROGRAMMABLE INTERVAL TIMER

1)What is PIT

The Programmable Interval Timer (PIT) chip basically consists of an oscillator, a prescaler and 3 independent frequency dividers. Each frequency divider has an output, which is used to allow the timer to control external circuitry.

2)Frequency Dividers

The basic principle of a frequency divider is to divide one frequency to obtain a slower frequency. This is typically done by using a counter. Each "pulse" from the input frequency causes the counter to be decreased, and when that counter has reached zero a pulse is generated on the output and the counter is reset.The PIT has only 16 bits that are used as frequency divider, which can represent the values from 0 to 65535. Since the frequency can't be divided by 0 in a sane way, many implementations use 0 to represent the value 65536.

3)Outputs

Channel 0 is connected directly to IRQ0, so it is best to use it only for purposes that should generate interrupts. Channel 1 is unusable. Channel 2 is connected to the PC speaker, but can be used for other purposes without producing audible speaker tones.

4)I/O Ports

I/O port	Usage
0x40	Channel 0 data port (read/write)
0x41	Channel 1 data port (read/write)
0x42	Channel 2 data port (read/write)
0x43	Mode/Command register (write only, a read is ignored)

5)Operating Modes

While each operating mode behaves differently, some things are common to all operating modes. This includes:

Initial Output State-Every time the mode/command register is written to, all internal logic in the selected PIT channel is reset, and the output immediately goes to its initial state (which depends on the mode).

Changing Reload Value-A new reload value can be written to a PIT channel's data port at any time. The operating mode determines the exact effect that this will have.

Current Counter-The current counter value is always either decremented or reset to the reload value on the falling edge of the (1.193182 MHz) input signal.

Current Counter Reload-In modes where the current count is decremented when it is reloaded, the current count is not decremented on the same input clock pulse as the reload – it starts decrementing on the next input clock pulse.

6)Usage

6.1)Implementing Sleep

The PIT's generating a hardware interrupt every n milliseconds allows you to create a simple timer.

6.2)Preemptive Multitasking

The timer IRQ can also be used to perform preemptive multitasking. To give the currently running task some time to run, set a threshold, for example of 3 ticks. Use a global variable like the one before but go up from 0, and when that variable hits 3, switch tasks.

7)References

7.1)https://wiki.osdev.org/Programmable_Interval_Timer#The_Oscillator

7.2)https://en.wikipedia.org/wiki/Programmable_interval_timer

CALLING GLOBAL CONSTRUCTORS

1)What are Global Constructors

Global Constructors are supposed to have run before your main function, which is why the program entry point is normally a function called _start. This function has the responsibility of parsing the command line arguments, initializing the standard library (memory allocation, signals, ...), running the global constructors and finally exit(main(argc, argv))

2)System V ABI

The System V ABI (as used by i686-elf-gcc, x86_64-elf-gcc, and other ELF platforms) specifies use of five different object files that together handle program initialization. These are traditionally called crt0.o, crti.o, crtbegin.o, crtend.o, and crtn.o. Together these object files implement two special functions: _init which runs the global constructors and other initialization tasks, and _fini that runs the global destructors and other termination tasks.

This scheme gives the compiler great control over program initialization and makes things easy for you, but you have to cooperate with the compiler, otherwise bad things will happen. Your cross-compiler will provide you with crtbegin.o and crtend.o. These files contain the internals that the compiler wishes to hide from you, but wants you to use. To get access to this information, you will need to provide your own implementation of crti.o and crtn.o.

3)Using crti.o, crtbegin.o, crtend.o, and crtn.o in a Kernel

The compiler always supplies crtbegin.o and crtend.o, but normally the C library supplies crti.o and crtn.o, but not in this case. The kernel should supply its own crti.o and crtn.o implementation (even if it would be otherwise identical to the user-space libc version). A kernel is linked with -nostdlib (which is the same as passing -nodefaultlibs and -nostartfiles) which disables the "start files" crt*.o that are normally automatically added to the link command line. By passing -nostartfiles, we promise to the compiler that we take on the responsibility ourselves to call the "program initialization tasks" in the crtbegin.o and crtend.o files. This means as we need to manually add crti.o, crtbegin.o, crtend.o, and crtn.o to the command line. Since we provide crti.o and crtn.o ourselves, that is trivial to add to the kernel command line. However, since crtbegin.o and crtend.o are installed inside a compiler-specific directory, we'll need to figure out the path.GCC offers an option just to do this. If i686-elf-gcc is your cross-compiler and $CFLAGS is the flags you would normally provide to your compiler.

4)Using crti.o, crtbegin.o, crtend.o, and crtn.o in User-Space

```
.section .init

.global _init

.type _init, @function

_init:
        push %rbp
        movq %rsp, %rbp
.section .fini
.global _fini
.type _fini, @function
_fini:
        push %rbp
        movq %rsp, %rbp
        .section .init
                popq %rbp
        ret

.section .fini
                popq %rbp
        ret
```

5)References

5.1)https://wiki.osdev.org/Calling_Global_Constructors

5.2)https://cs.stackexchange.com/questions/101732/in-a-kernel-what-are-global-constructors

PAGING

1) What is Paging?

Paging is a system which allows each process to see a full virtual address space, without actually requiring the full amount of physical memory to be available or present. 32-bit x86 processors support 32-bit virtual addresses and 4-GiB virtual address spaces.n addition to this, paging introduces the benefit of page-level protection. In this system, user processes can only see and modify data which is paged in on their own address space, providing hardware-based isolation. System pages are also protected from user processes. On the x86-64 architecture, page-level protection now completely supersedes Segmentation as the memory protection mechanism. On the IA-32 architecture, both paging and segmentation exist.

2) 32-bit paging (Protected Mode)

2.1)MMU

Paging is achieved through the use of the Memory Management Unit (MMU). On the x86, the MMU maps memory through a series of tables, two to be exact. They are the paging directory (PD), and the paging table (PT).

Both tables contain 1024 4-byte entries, making them 4 KiB each. In the page directory, each entry points to a page table. In the page table, each entry points to a 4 KiB physical page frame. Additionally, each entry has bits controlling access protection and caching features of the structure to which it points. The entire system consisting of a page directory and page tables represents a linear 4-GiB virtual memory map.

Translation of a virtual address into a physical address first involves dividing the virtual address into three parts: the most significant 10 bits (bits 22-31) specify the index of the page directory entry, the next 10 bits (bits 12-21) specify the index of the page table entry, and the least significant 12 bits (bits 0-11) specify the page offset. The then MMU walks through the paging structures, starting with the page directory, and uses the page directory entry to locate the page table. The page table entry is used to locate the base address of the physical page frame, and the page offset is added to the physical base address to produce the physical address.

2.2)Page Directory

The topmost paging structure is the page directory. It is essentially an array of page directory entries.

When PS=0, the page table address field represents the physical address of the page table that manages the four megabytes at that point. Please note that it is very important that this address be 4-KiB aligned. This is needed, due to the fact that the last 12 bits of the 32-bit value are overwritten by access bits and such. Similarly, when PS=1, the address must be 4-MiB aligned.

PAT, or Page Attribute Table. If PAT is supported, then PAT along with PCD and PWT shall indicate the memory caching type. Otherwise, it is reserved and must be set to 0.

G, or 'Global tells the processor not to invalidate the TLB entry corresponding to the page upon a MOV to CR3 instruction. Bit 7 (PGE) in CR4 must be set to enable global pages.

PS, or 'Page Size' stores the page size for that specific entry. If the bit is set, then the PDE maps to a page that is 4 MiB in size. Otherwise, it maps to a 4 KiB page table. Please note that 4-MiB pages require PSE to be enabled.

D, or 'Dirty' is used to determine whether a page has been written to.

A, or 'Accessed' is used to discover whether a PDE or PTE was read during virtual address translation. If it has, then the bit is set, otherwise, it is not. Note that, this bit will not be cleared by the CPU, so that burden falls on the OS =.

PCD, is the 'Cache Disable' bit. If the bit is set, the page will not be cached. Otherwise, it will be.

PWT, controls Write-Through' abilities of the page. If the bit is set, write-through caching is enabled. If not, then write-back is enabled instead.

U/S, the 'User/Supervisor' bit, controls access to the page based on privilege level. If the bit is set, then the page may be accessed by all; if the bit is not set, however, only the supervisor can access it. For a page directory entry, the user bit controls access to all the pages referenced by the page directory entry. Therefore if you wish to make a page a user page, you must set the user bit in the relevant page directory entry as well as the page table entry.

R/W, the 'Read/Write' permissions flag. If the bit is set, the page is read/write. Otherwise when it is not set, the page is read-only. The WP bit in CR0 determines if this is only applied to userland, always giving the kernel write access (the default) or both userland and the kernel =.

P, or 'Present'. If the bit is set, the page is actually in physical memory at the moment. For example, when a page is swapped out, it is not in physical memory and therefore not 'Present'. If a page is called, but not present, a page fault will occur, and the OS should handle it.

2.3)Page Table

In each page table, as it is, there are also 1024 entries. These are called page table entries, and are very similar to page directory entries.The first item, is once again, a 4-KiB aligned physical address. Unlike previously, however, the address is not that of a page table, but instead a 4 KiB block of physical memory that is then mapped to that location in the page table and directory. Note that the PAT bit is bit 7 instead of bit 12 as in the 4 MiB PDE.

2.4)CODE

```
mov eax, 0x0
mov ebx, 0x100000
.fill_table:
    mov ecx, ebx
    or ecx, 3
    mov [table_768+eax*4], ecx
    add ebx, 4096
    inc eax
    cmp eax, 1024
    je .end
    jmp .fill_table
.end:
```

3)References

3.1)https://wiki.osdev.org/Paging

3.2)https://en.wikipedia.org/wiki/Memory_paging