`Hello World` boot loader in Intel Assembly “hello.asm”

-------------------------------------------------------------------------

org 07c00h **i**

mov ax, cs **ii**

mov ds, ax **iii**

mov es, ax **iv**

call DispStr **v**

jmp $ **vi**

DispStr:

mov ax, BootMessage **vii**

mov bp, ax **viii**

mov cx, 16 **ix**

mov ax, 01301h **x**

mov bx, 000ch **xi**

mov dl, 0 **xii**

int 10h **xiii**

ret **xiv**

BootMessage: db “Hello, OS world!” **xv**

times 510-($-$$) db 0 **xvi**

dw 0xaa55 **xvii**

Explanation

----------------

**i** :- It is where you have an assembler and linker in one step. The org tells the assembler which tells the linker (in this case the same program) where in physical memory space to put the code that follows. Otherwise it assumes that your code would be located in the memory beginning from the offset 0 in you code segment. 7c00 is the physical address of the boot-loader.

**viii** :- es:bp = offset of string -> pointer to BootMessage string

**ix** :- talks about string length

**x** :- ax = 0x1301 -> ah = 0x13 and al = 0x01. This described as al = write mode ->1

**xi** :- bx = 0x000c -> bh = 0x00 and bl = 0x0c. This described as bh = page number -> 0 and bl = color -> 0x0c = light red

**xii** :- dl = 0x00 dh = 0x00. This described as dh = row -> 0 and dl = column -> 0

**xvi** :- $$ refers start of the section, which is 0x7c00; $ refers to start of the “times” line 0x7c**ZY**, where ‘ZY’ can’t be more than 510 bytes. This means indirectly N + (510 – N) = 510

**xvii** :- Continuing from the above description, we reserve a word 2 bytes for 0Xaa55 magic boot signature. This way, we can have a total of 512 byte block.

Transferring the above code to binary file

-------------------------------------------------------

nasm hello.asm –f bin –o hello.bin

Running the code in Qemu emulator

-------------------------------------------------

qemu-system-i386 hello.bin

-----------------------------------------------------------------------------------------------------------------------------------------

**16-bit Real Mode** :- CPU manufacturers must go to great lengths to keep their CPUs (i.e their specific instruction set) compatible with earlier CPUs, so that older software, and in particular older OS, can still run on them most modern CPUs. Intel solution will be creation of family **X86**, which had support for 16-bit instructions and no notation of memory protection.

Q.N: - Forgetting compatibility, Can we boot with direct 32-bit while protected mode enabled?

What does it mean by CPU is 16-bit? : - if it can accept a maximum of 16-bit at once (like adding two 16-bit numbers), what if overflow occurs or 17-bit created during calculation?

------------------------------------------------------------------------------------------------------------------------------------------

Questions

1. In os-dev.pdf at page 4 shows a machine code boot sector, with each byte displayed in hexadecimal and the same too for hello.bin file by using `**od –t x1 –A n hello.bin**` command. But I observed the one which is described on the pdf contains exactly 512 byte but hello.bin 768 byte. How could this code boot from qemu emulator?
2. How the offset increases by 1 byte when I try to display some character from assembly code after 0x7C00? Check page 19 in os-dev.pdf. and if we didn’t say org 0x7c00, how is that possible that the correct offset is base + the\_secret, which offset is 0x7c00 in our case

-----------------------------------------------------------------------------------------------------------------------------------------

How to communicate with the screen and how to ask BIOS to do that action

----------------------------------------------------------------------------------------------------

Somewhere in memory of the computer there will be some BIOS machine code that knows how to write to the screen. We could possibly find the BIOS code in memory and execute it somehow, but this might prone to errors when there are differences between BIOS routine internals on different machines.

We can make use of a fundamental mechanism of the computer: **interrupts**. Each interrupts is represented by a unique number that is an index to the interrupt vector, a table initially setup by BIOS at the start of memory (at physical address 0x0) that contains address pointers to interrupt service routines (ISRs). BIOS multiplexes the IRS by what we could imagine as a big ‘switch’ statement, based usually on the value set in one of the CPUs general purpose registers, ax, prior to raising the interrupt. *To display some character by using BIOS routines*, we need interrupt 0x10 and to set ah to 0x0e (to indicate tel-type mode) and al to the ASCII code of the character we wish to print

Real Mode

--------------

When the CPU is in 16-bit real mode, the highest address we can refer is 64KB. To get around this problem, the CPU designers added a few more registers called segment registers. We can imagine main memory as being divided into **segments** that are indexed by the segment registers.

The most confusing thing about segment addressing is that adjacent segments overlap almost completely but for 16 bytes, so different segment and offset combinations can actually point to the same physical address. We calculate address by multiplying segment register value by 16 and add offset inside ax.

Because we do not use the org directive, the assembler doesn’t offset our labels to correct memory locations when the code is loaded by BIOS to address 0x7c00, so the first attempt to print an ‘x’ will fail. However, if we set the data segment registers to 0x7c0, the CPU will do this offset for us (i.e. 0x7c0 \* 16 + the\_secret) and so the second attempt will correct print the ‘x’. Instead explicitly state to the CPU which segment register to use when computing the physical address, using instead the general purpose segment registers (es).

Note: - we can’t store address directly to segment register.

Segment-based addressing allows us to reach further into memory, upto a little over 1Mb (0XFFFF \* 16 + 0xFFFF).

Lower memory layout after boot

-------------------------------------------

|  |  |  |
| --- | --- | --- |
| 0x100000 | FREE | 1MB |
| 0XC0000 | BIOS (256 KB) | 765K |
| 0XA0000 | Video Memory (128 KB) | 640K |
| 0X9FC00 | Extended BIOS Data Area (639 KB) | 639K |
| 0X7E00 | FREE (638 KB) | 31.5K |
| 0X7C00 | Loaded Boot Sector (512 Bytes) | 31K |
| 0X500 |  | 1.25K |
| 0X400 | BIOS Data Area (256 Bytes) | 1KB |
| 0X0 | Interrupt Vector Table (1KB) | 0 |

Accessing Hard Disk

---------------------------

Most hard disk devices have more functionality automated on local chips, but again the bus technologies with which such devices connect to the CPU affects how we access them. Thankfully, BIOS can offer a few disk routines that abstract all of these differences for common disk devices.

The specific BIOS routine we are interested in here is accessed by raising interrupt 0x13 after setting the register al to 0x02.

For one reason or another (e.g. we indexed a sector beyond the limit of the disk, an attempt was made to read a faulty sector, the floppy disk was not inserted into drive … etc), BIOS may fail to read the disk for us, so it is important to know how to detect this. BIOS updates some registers to let us know what happened, carry flag (CF) of the special flags register. It set to signal a general fault, and **al** is set to the number of sectors actually read, as opposed to the number requested.

Being able to read more data from the disk will be essential for boot-strapping our operating system.

Entering 32-bit protected mode

------------------------------------------

Main difference

* Registers are extended to 32-bits, with their full capacity
* Additional two segment registers (FS and GS) will be used
* 32-bit memory offset will be available
* Code in one segment can be prohibited from executing code in a more privileged segment
* CPU can implement virtual memory for user processes, such that pages of a process’s memory can be swapped transparently between the disk and memory on an as-needed basis.
* Interrupt handling is also more sophisticated
* We prepare a complex data structure in memory called the global descriptor table (GDT), which defines memory segment and their protected-mode attributes.
* We have a special instruction to load GDT into the CPU, before setting a single bit in a special CPU control register to make the actual switch
* GDT is defined in assembly language, but sadly this low-level switch-over is avoidable if we wish later to load a kernel that has been compiled from **c**.
* *We can no longer use BIOS* once switched into 32-bit protected mode.

Adapting to life without BIOS

* BIOS routines, having been coded to work only in 16-bit real mode, are no longer valid in 32-bit protected mode. This means is that a 32-bit operating system must provide its own drivers for all hard ware of the machine (e.g. keyboard, screen, disk drive … etc)

How to Display message to the screen

* Display device can be configured into one of several resolutions in one of two modes, text mode and graphics mode; and that what is displayed on the screen is a visual representation of specific range of memory.
* We need to set its ASCII code and attribute at the correct memory address for the current VGA mode, which is usually at address 0xb8000
* Note that, although the screen is displayed as columns and rows, the video memory is simply sequential. For example, the address of the column 5 on row 3 can be calculated as follows: 0xb8000 + 2 \* (row \* 80 + col)
* Check figure 4.1 for better understanding

Q.N: As you can see from the code, edx contains string memory address, so shall we first point edx to the address or if not how edx knows about string address?

Understanding GDT

--------------------------

A segment register becomes an index to a particular segment descriptor (SD) in the GDT. Segment descriptor contains base address, segment limit and various flags which affect how the CPU interprets the segment, such as the privilege level of code that runs with in it or whether it is read-or write-only.

**Basic flat model**: - the simplest workable configuration of segment registers described. Where by two over lapping segments are defined that cover the full 4GB of addressable memory, one for code and the other for data. In this model these two segments have no protection from each other, nor is there any attempt to use the paging features.

|  |
| --- |
| GDT |
| Gdt\_data |
| Gdt\_code |
| Null\_descriptor |

**Null Descriptor**: - is a simple mechanism to catch mistakes where we forget to set a particular segment register before accessing an address, which is easily done if we had some segment registers set to 0x0 and forget to update them to the appropriate segment descriptors after switching to protected mode.

Defining GDT in assembly

----------------------------------

The CPU needs to know how long our GDT is; we don’t actually directly give the CPU the start address of our GDT but instead give it GDT descriptor (i.e. something that describes the GDT). It is a 6-byte structure containing GDT size (16-bits) and GDT address (32-bit)

Q.N: - (gdt\_end – gdt\_start – 1) Size of our GDT, always less one of the true size.

|  |  |
| --- | --- |
| value | size |
| identical with GDT code in type flags  GDT data |  |
| \* |  |
| 11001111 b | 1 B |
| 1001101010 b | 1 B |
| 0x0  GDT code | 1 B |
| 0x0 | 2 B |
| 0xFFFF | 2 B |
| 0x0  Null Descriptor | 4 B |
| 0x0 | 4 B |

Making the Switch

-------------------------

Once both the GDT and the GDT descriptor have been prepared with in our **both sector**, we are ready to instruct the CPU to switch from 16-bit real-time into 32-bit protected mode.

Q.N: - I don’t see the GDT structure is enough to manage 32-bit protected mode memory management. Explain GDT from start to end in doing those jobs of memory accessing.

Switch: - 1. Disable interrupt 2.Tell the CPU about the GDT we just prepared 3. Start 32-bit protected mode by enabling CR0 first bit 4. After switching CPU mode, there is a risk that the CPU may process some stages of an instruction’s execution in the wrong mode. The simple solution will be to force the CPU to finish any jobs in its pipeline, so that we can be confident that all future instructions will be executed in the correct mode. 5. Jmp <segment> : < address offset> since our code segment will not be valid in protected mode, we must update our CS register to the offset of the code segment descriptor of our GDT.

Note

------

**Pipelining** (A modern CPU process different stages of an instruction’s execution in parallel, single CPUs as opposed to parallel CPUs)

**Far jump**: - it will automatically cause the CPU to update our CS register to the target segment. If this concept works for DS, how does the CPU differentiate those two cases? If we refer the same code descriptor base address for all case, is this will be enough for huge program or combination of programs?

**[bits 32]**: - directive to tell our assembler that from that point onwards, it should encode in 32-bit mode instructions.

Putting it all together

----------------------------

* First understand (4.3 and 4.4 assembly) points**!**

Stack

-------

Used as temporary storage for local variables, argument passing and most importantly in 16-bit mode, the stack works only on 16-bit boundaries.

Q.N:- based on the above point, the stack boundaries changes to 32-bit or 64-bit when it works with 32-bit or 64-bit mode?

Function

------------

The callee function should know **where to return** after our function has been called. Solution 1: The caller could store the correct return address (i.e the address immediately after the call) in some well-known location, then the called function jump back to that stored address. Solution 2: The CPU keeps track of the current instruction being executed in the special register IP (Instruction pointer), which sadly, we can’t access directly. However using **call** and **ret**, which do exactly what we want: call behaves like **jmp** but additionally, before actually jumping, pushing the return address onto the stack; ret then POPs the return address off the stack and jumps to it.

NOTES: - Storing values inside register is not safe especially during calling function and expecting to get original values from register after return. That is why we save register values to stack and then pop them off again. (i.e. restore the registers original values immediately before it returns). pusha and popa :- all registers to and from the stack respectively will play a great role in doing the explained objectives.

Using BIOS to Read the Disk

-------------------------------------

mov ah, 0x02 ; BIOS read sector function

mov dl, 0 ; Read drive 0 (i.e. first floppy drive)

mov ch, 3 ; Select Cylinder 3

mov dh, 1 ; Select the track on 2nd side of floppy disk, since this count has a base of 0

mov cl, 4 ; Select the 4th sector on the track, not the 5th, since this has a base of 1

mov al, 5 ; Read 5 sectors from the start point

; Lastly, set the address that we’d like BIOS to read the sectors to, which BIOS expects to find in ES : BX

; (i.e. segment ES with offset BX).

mov bx, 0xa000 ; Indirectly set ES to 0xa000

mov es, bx

mov bx, 0x1234 ; Set BX to 0x1234

; In our case, data will be read to 0xa000 : 0x1234, which the CPU will translate to physical address

; 0xa1234

int 0x13 ; Now issue the BIOS interrupt to do the actual read.

Writing, Building and Loading you kernel

------------------------------------------------------

Note: assembly have a draw back in architecture dependency

Note: The idea of using high level language compiler is to map higher level constructs, such as control structures and function calls onto assembly template code, and so the downside is, the generic templates may not always be optional for specific functionality.

**gcc –ffreestanding –c basic.c –o basic.o**

Note: - ffreestanding : assert that compilation takes place in a free standing environment. This Implies –fno-builtin. A free standing environment is one in which the standard library may not exit, and program startup may not necessarily be at “main”. The most obvious example is an OS kernel. This is equivalent to –fno-hosted.

Note: - Changing .c file in .o object file has an advantage in easily relocating into a large binary file when linked in with routines from other routines in other libraries, since code in the object file uses **relative rather than absolute** internal memory reference.

Try “objdump –d basic.o” to see object files contents.

The linker will link all of the routines described in the input object files into one executable binary file, effectively stitching them together and converting those relative addresses into absolute address within the aggregated machine code. E.g. call <function\_x\_lable> will become call 0x2345, where 0x12345 is the offset within the output file. In short, the linker will converted our annotated machine code file into a row machine code file.

Note: - To output row machine code into a file basic.bin, we can use “**ld –o basic.bin –Text 0x0 –oformat binary basic.o**” command. –Ttext 0x0 works in the same way as the org directive we used in our earlier assembly routines, by allowing us to tell the compiler to offset label addresses in our code to their absolute memory addresses when later loaded to a specific origin in memory.

Note: - Another benefit of understanding assembly helps to understand reverse-engineer of any software by disassembling. Problem of disassembling machine code is that some of those bytes may have been reserved as a data but will show up as assembly instructions. E.g. **ndisasm –b 32 basic.bin > basic.dis**, the –b 32 simply tells the disassemble to decode to 32-bit assembly instructions.

Note: - “mov [ebp – 0x4], 0xbaba”: - is calculated on the fly by the CPU based on the current address of register ebp. It is not pre-processing; it forms a part of the CPU instruction. With this form of addressing the CPU is allowing us to do more per instruction cycle. OR mov eax, ebp -> sub eax, 0x4 -> mov [eax], 0xbaba.

Note: -

void caller\_fuction () { | int callee\_function (int my\_arg){

callee\_function (0xdede); } | return my\_arg; }

Disassemble the above code after converting to .bin file type. It will give the below list

.1 push ebp

.2 mov ebp, esp

.3 sub esp, byte +0x8

.4 mov dword [esp], 0xdede

.5 call dword, 0x14

.6 leave

.7 ret

.8 push ebp

.9 mov ebp, esp

.10 mov eax, [ebp + 0x8]

.11 pop ebp

.12 ret

Q.N point 10 mov eax, [ebp + 0x8], which ebp caller or callee? Page 47, because the caller put that value onto the top of their stack, then we put our stack base at the top of their stack to establish our stack frame.

Q.N: - page 45 (sub esp, byte + 0x10) decrements current stack with 16 byte why not 8byte if the reason is to occupy for optimization reason. Why 16 bytes? It is more than the necessary space

Executing our kernel code

-----------------------------------

INVOLVED STEPS ARE **1**. Write and Compile the kernel code **2**. Write and assemble the boot sector code **3**. Create a kernel image that includes not only our boot sector but our compiled kernel code **4**. Load our kernel code into memory (kernel code or kernel image) **5**. Switch to 32-bit protected mode **6**. Begin executing our kernel code

1 void main(){

char\* video\_memory = (char \*) 0xb8000;

\*video\_memory = ‘x’; }

gcc –ffreestanding –c kernel.c –o kernel.o

ld –o kernel.bin –Ttext 0x1000 kernel.o –oformat binary

Now we tell the linker that the origin of our code once we load it into memory will be 0x1000.

2. Note: lack of BIOS will make it hard for us to use the disk. We would have to write a floppy or hard-disk driver ourselves! To simplify the problem of which disk and from which sectors to load the kernel code, the boot sector and kernel of an operating system can be grafted together into a kernel image, which can be written to the initial sectors of the boot disk, such that the boot sector code is always at the head of the kernel image. Once we have compiled the boot sector, we can create our kernel image with ` cat boot\_sect.bin kernel.bin > os\_image ` command.

Note: BIOS stores our **boot drive in DL**, so it’s best to remember this for later

Finding OUR way into the Kernel

-------------------------------------------

The C compiler can decide to place code and data wherever it chooses in the output file. Entering our kernel in the correct place is too dependent upon the ordering of elements (e.g. functions) in our kernel’s source code and upon the whims of the compiler and linker, so we need to make this more robust.

A simple trick will be writing very simple assembly routine that is always attached to the start of the kernel machine code and whose sole purpose is to call the entry function of the kernel.

E.g. `**main**` doesn’t exist as a label within this code, since it is expected to exist within one of the other object files, this expectance is expressed by the directive [**extern main**], at the top of the file, and the linker will fail if it doesn’t find such a label.

We must compile it to preserve Meta information about the labels it must resolve. ` nasm kernel\_entry.asm –f elf –o kernel\_entry.o `. The option –f elf tells the assembler to output an ELF, is also the default format output by C compiler.

Now rather than simple linking the kernel.o file with itself to create kernel.bin, we can link it with kernel\_entry.o, as follows `ld –o kernel.bin –Ttext 0x1000 kernel\_entry.o kernel.o --oformat binary`

We can construct our kernel image file with the following command `cat boot\_sect.bin kernel.bin > os-image`

Developing Essential Device Drivers and a File-System (Check the book for this topic)

………………………………………………………………………………………………………………………..  
…………………………………………………………………………………………………………………………………………

……………………………………………………………………………………………………………………………………………………………

Extra Notes

---------------

**Stack Growth**: - The behavior of stack (growing up or growing down) depends on the application binary interface (ABI) and how the call stack (aka activation record) is organized.

**Object Oriented** Concept: - An object is a software bundle of related state and behavior. The prime purpose of C++ programming was to add object orientation to the c programming language. To create an object in code, that has certain properties and methods.

Nasm key word (**lgdt** and **lidt**): - both take a 6-byte memory area as an operand: they load a 32-bit linear address and a 16-bit size limit from that area (in the opposite order) into the ‘GDTR’ or ‘IDTR’ (interrupt descriptor table register). These are the only instructions which directly use linear address, rather than segment / offset pairs. How lgdt finds GDTR?

**leave**: - instruction is an alternative to the following steps, that restore the original stack of the caller, reciprocal of the first two instruction of the following. **mov esp, ebp** -> **pop ebp**

**Variables**: - are simply references to allocated memory space.

**lodsb** : - loads a byte from `[DS:SI]` or `[DS:ESI]` into `AL`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` or `ESI`.

**[SECTION signature start=0x7dfe]** == times 510 – ($ - $$) db 0

**ALIGN:** - macros provides a convenient way to align code or data on a word

**The** **linker** takes object files as inputs, then joins them together, but resolves any labels to their correct addresses.

**Extern** in assembly: - it is used to declare a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module and needs to be referred by this one.

List of Questions

1. Discover how to link different language object file with c object file and result one executable.
2. Even if we make Kernel Image to skip writing driver for that matter, how does the image itself help each other to boot strapping process by understanding how to read from the device?
3. I see no segment address in GDT or I don’t see how it works to point or locate the protected mode segments

**AIM OF THIS DOCUMENT**

* To create bootable shell which will work like operating system and able to manage running other processes like a single job.
* Convert single-process to multi-process.