Contents

1 OS Development quick notes

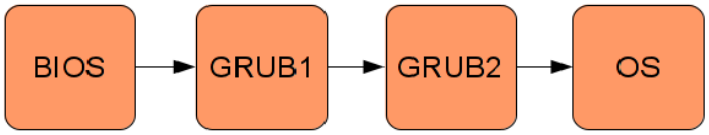
Quick notes I'll take while reading **The little book about OS Dev**.

1.1 Resources used

- The little book about os dev
- Makigas: construyendo un sistema operativo (youtube)
- wiki.osdev.org
- Andreas Kling code tour videos

1.2 Basics

1.2.1 Booting



When the computer boots up, first it loads the **POST** to check that all the hardware is healthy and able to work, if it fails, normally is when the "beeps" from the speaker gives information about what could be failing. After this, it starts the BIOS (or on modern days, UEFI), this is basically the firmware, a program that (normally) is located on a ROM in the motherboard responsible for initializing devices such as memory, searches for bootable devices (hard disk, dvd, . . .) and then it passes the control to the bootloader, the bootloader is responsible for loading the actual operating system.

1. BIOS

   On BIOS firmwares, it loads the bootloader from the first sector in the MBR (Master Boot Record) of a bootable device

2. UEFI

   On UEFI firmwares, the bootloader is loaded from the **EFI** partition (EFI System Partition).

   A bootloader is a lowlevel complex program, while I could create one from scratch I for now will prefer to use a standard bootloader such as GRUB

1.3 Loader

The first "OS" example that the book shows to compile is a simple one that simply will load `0xCAFEBABE` into register **eax**.

To do this, it writes assembly code (we can't use C without a stack set up)

```
global loader  ; entry symbol for the program to load
; constants definitions
MAGIC_NUMBER equ 0x1BADB002  ; THis is the required header parameters used for
FLAGS equ 0x0
CHECKSUM equ -MAGIC_NUMBER

section .text:
align 4  ; align variables into memory locations that are divisible by 4, this

; variable definitions
dd MAGIC_NUMBER
dd FLAGS
dd CHECKSUM
```

```
loader:
        mov eax, 0xCAFEBABE

.loop:
        jmp .loop
```

In this assembly code we just define the correct header so it can be used by the standard multiboot specification.

This is then compiled into a 32bit ELF object file

```
nasm -f elf32 loader.s
```

And finally linked to have an executable file.

```
ENTRY(loader) /* the name of the entry label */
SECTIONS {
. = 0x00100000; /* the code should be loaded at 1 MB */
.text ALIGN (0x1000) : /* align at 4 KB */
{
*(.text) /* all text sections from all files */
}
.rodata ALIGN (0x1000) : /* align at 4 KB */
{
*(.rodata*) /* all read-only data sections from all files */
}
.data ALIGN (0x1000) : /* align at 4 KB */
{
*(.data) /* all data sections from all files */
}
.bss ALIGN (0x1000) : /* align at 4 KB */
{
*(COMMON) /* all COMMON sections from all files */
*(.bss) /* all bss sections from all files */
}
}
```

THen we link the object file compiled before with the link script so we can generate an executable.

```
ld -T link.ld -melf_i386 loader.o -o kernel.elf
```

Now I have an executable elf file, **kernel.elf**, this is the kernel to be loaded, we need a bootloader to execute this, in the book they show how to make an ISO that GRUB can boot, that's what I did to test, however, Andreas Kling, author of SerenityOS, showed how to make QEMU work as it's own bootloader, by passing the `-kernel` parameter in the CLI, that saved me some time, I won't need to create an ISO each time I want to test the OS.

```
qemu-system-i386 -d cpu -kernel kernel.elf
```

The `-d` option just indicates what I will want to show in the debug logs, with indicating cpu I can see a list of the registers.

So after booting, I can se that the **eax** register is correctly set to **0xBADC0FFE**

```
CCS=00000004 CCD=00950000 CCO=EFLAGS
EFER=0000000000000000
EAX=badc0ffe EBX=00009500 ECX=0010000c EDX=00010511
ESI=00000000 EDI=00001000 EBP=00000000 ESP=00006f08
EIP=00100011 EFL=00000006 [-----P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     000cb2b4 00000027
IDT=     00000000 000003ff
CR0=00000011 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
CCS=00000004 CCD=00950000 CCO=EFLAGS
EFER=0000000000000000
```

1.3.1 Getting to C

We don't want to use assembly all the time unless you have some kind of kink, so we will set up the correct enviroment to use C instead, for this we need to set up a memory **stack** which C will use.

To do this, we need to change where the **esp** register points to, but we can't just randomly use any address (we could, but it's a risk), what we can

do is to reserve some memory space by declaring data in the .bss section of our ELF file (If compiling to an ELF, in other way I guess you would need to do some hacky thing in the loader logic?)

To set up a stack, we need simply need to reserve uniniatilaized space on memory, then make SP point to the end of that space (the stack grows downwards, so it should start at a high address), to do this, the book says to create the constant in the loader, then reserve that size in in the .bss section (a section used to declare uninitalized variables, (I think?)) with the pseudo-instruction **resb** which will reserve the number of bytes passed as argument, so we then we can just make the stack pointer register set to the tag used in the .bss section + the stack size constant defined.

```
STACK_SIZE equ 4096

loader:
    mov esp, kernel_stack + STACK_SIZE

section .bss

        align 4
        kernel_stack:
        resb STACK_SIZE
```

Now that we have a stack we can use C code in our kernel, to do this we need to be able to call C functions from assembly, (I guess it could be any compiled language? Just the process would be different in the assembly part I guess)

The process to call external functions is by using the **extern** assembly keyword and indicate the name of the function we are going to use, however firstly we need to link the object output of our C kernel with the object output of our assembly loader, this is already expected in the linker script done before but now we need to add the C object output into our **ld** command, and since C compilation command is also needed, it's convienent to have a Makefile as suggested by the book. And after indicating this, we can just use the instruction **call**, and parameters should be pushed into the stack before, the book uses the *cdecl* calling convention, why? No idea, but after reading Wikipedia, the `cdecl` calling convention uses RTL right to left order, so the last parameter will be the first argument for the function, integer values and

memory addresses will be put into **EAX** register, what about characters and strings? No idea for now, but I supposse since characters are just numbers it will be passed into the EAX register too? Floating values are other story that I dont wanna know right now :D

So a simple example would be:

hehe.c:

```
int sum(int n1,int n2){
  return n1 + n2;
}
```

loader.s

```
extern sum
;...
push 1
push 2
call sum
```

This would be like executing in C

```
sum(2,1);
```

And then in the EAX register we can see the number 00000003

1. THE MAKEFILE

```
LDFLAGS = -T link.ld -melf_i386
OBJS = loader.o kmain.o
AS = nasm
ASFLAGS = -f elf32
CC = gcc
CFLAGS = -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles
SRC = src/*.c

all: kernel.elf

build: loader
        $(CC) $(CFLAGS) -c $(SRC)  -o kmain.o
```

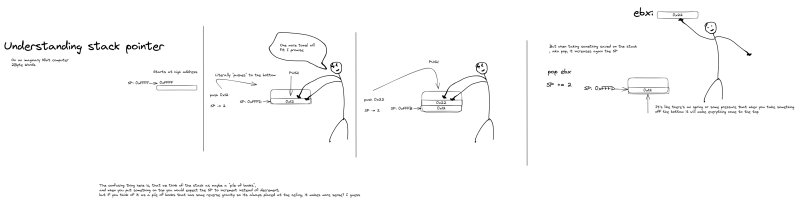
```
clean:
        rm -rf *.o *.elf
$(OBJS): build

loader: loader.s
        $(AS) $(ASFLAGS) loader.s

kernel.elf: $(OBJS)
        ld $(LDFLAGS) $(OBJS) -o kernel.elf

run:
        qemu-system-i386 -d cpu -D log -kernel kernel.elf
```

It's probably not the best Makefile ever done but make is one of those thing I learn when I need to use it and then just forget about it xD

2. How the stack pointer works

   Sidenotes created because I had a bad understanding of how the stack pointer works since years ago xD

   :ID: cd0b55c9-a642-4bf7-bb04-9b7764bceb68



   draw: `https://excalidraw.com/#json=oV1iOTDce_1U9P0qcz8Kq,_89CUPrnoM7dZX3rNo0jxg`

   IMPORTANT NOTE: While in the drawing the values in the stack "dissapear" from the stack when reading them with **pop**, in reality when you use pop the stack pointer is modified, but the data in the stack is **NOT** removed, it will still be there.

   Example given by allison on the osdev discord guild:

	A	B	C	D	E	F	G	H	I	J	K	L
1	address		nothing on stack		push 0x12		push 0x34		push 0x56		pop	
2												
3	512	200		<-SP								
4	511	1FF			0x12	<-SP	0x12		0x12		0x12	
5	510	1FE					0x34	<-SP	0x34		0x34	<-SP
6	509	1FD							0x56	<-SP		
7	508	1FC										
8	507	1FB										
9	506	1FA										
10	505	1F9										
11	504	1F8										
12	503	1F7										
13												

1.3.2 Drivers

I WANT A HELLO WORLD AND I WANT IT RIGHT NOW!

The first driver we are going to write is one to print text into the screen, since a driver is a program that acts as a layer to communicate hardware and kernel, I will put this part of writing to screen in this section.

2 Assembly notes

I know some basic assembly, so I will take notes of those instructions, directives or whatever I dont know so I can remember it correctly

2.1 DD

The **dd** (defined double world, a word normally is 2 Bytes, so double is 4) is used to define a variable as a 4Byte value.

2.2 EQU

***equ** directive is used to define constant values, for example

```
ZERO equ 0x0 ; will declare ZERO as 0x0
```