

SOLAB2 : MIT JOS lab 1

booting a PC

PARTE 1a (esercizi 1-7)

re@di.unimi.it

SOLAB2 : MIT JOS lab 1

booting a PC

Operazioni preliminari I

```
cd /mnt  
sudo mkdir joslabs  
sudo chown -R user joslabs  
sudo chgrp -R user joslabs  
sudo mount /dev/sda1 /mnt/joslabs  
cd joslabs  
cd lab
```

SOLAB2 : MIT JOS lab 1

booting a PC

Operazioni preliminari II

```
cd conf  
vi env.mk
```

Decommentare la riga #QEMU e modificare come segue:

```
QEMU=/opt/mitqemu/bin/qemu
```

```
salvare...
```

SOLAB2 : MIT JOS lab 1

booting a PC

Operazioni preliminari III

```
cd
```

```
echo 'add-auto-load-safe-path /mnt/joslabs/lab/.gdbinit' >  
.gdbinit
```

SOLAB2 : MIT JOS lab 1

booting a PC

Compilare il kernel JOS

```
cd /mnt/joslabs/lab  
make
```

Questo comando genera `obj/kern/kernel.img` che contiene il boot loader (`obj/boot/boot`) e il kernel (`obj/kernel`)

```
make qemu
```

L'ultimo comando utilizzato vi porta nel kernel monitor. Per uscire **ctrl+C**

SOLAB2 : MIT JOS lab 1

booting a PC

Eseguire il kernel JOS (II)

```
K> help
```

```
help - display this list of commands
```

```
kerninfo - display information about the kernel
```

```
K> kerninfo
```

```
Special kernel symbols:
```

```
entry  f010000c (virt)  0010000c (phys)
```

```
etext  f0101a75 (virt)  00101a75 (phys)
```

```
edata  f0112300 (virt)  00112300 (phys)
```

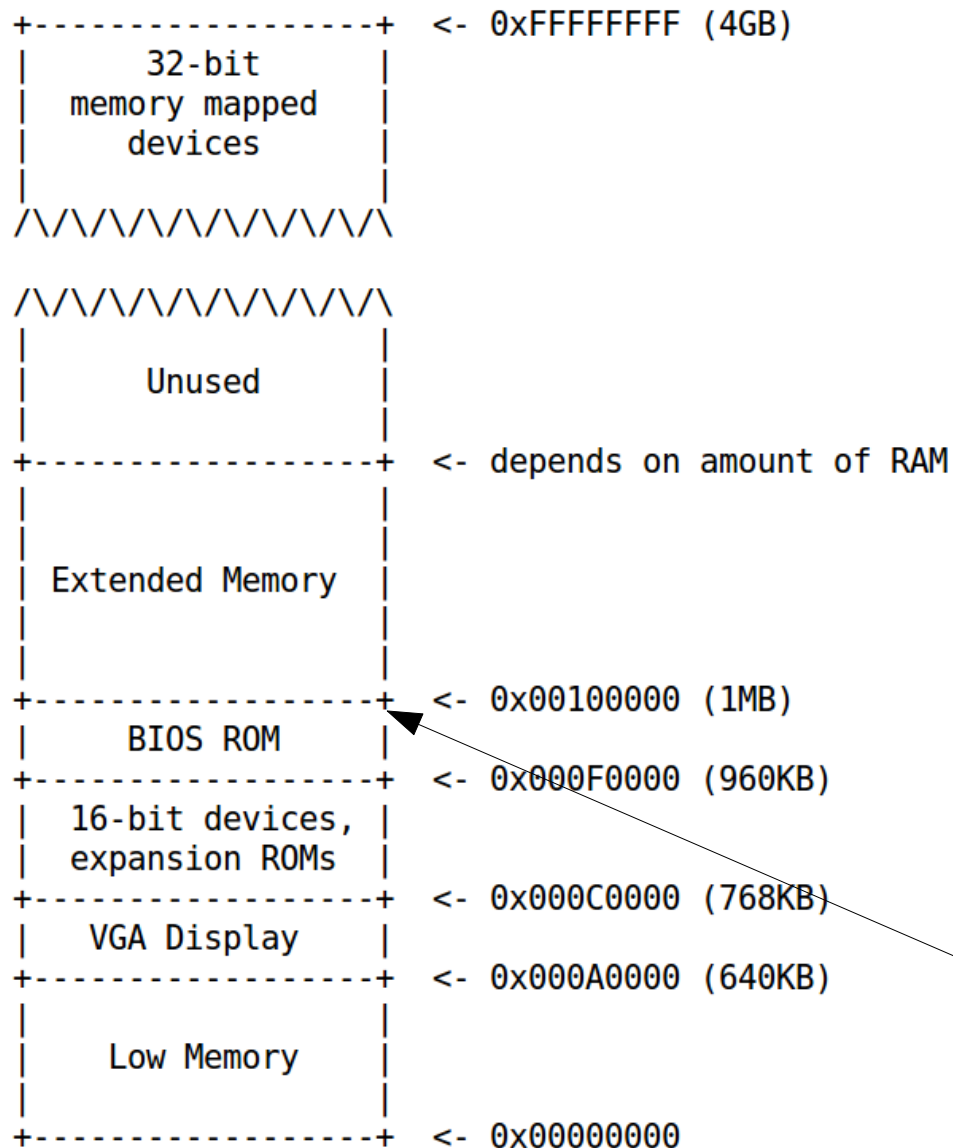
```
end    f0112960 (virt)  00112960 (phys)
```

```
Kernel executable memory footprint: 75KB
```

```
K>
```

SOLAB2 : MIT JOS lab 1

booting a PC



Spazio degli indirizzi fisici di un PC

Processore Intel 8088 può indirizzare 1 Mb di memoria fisica.

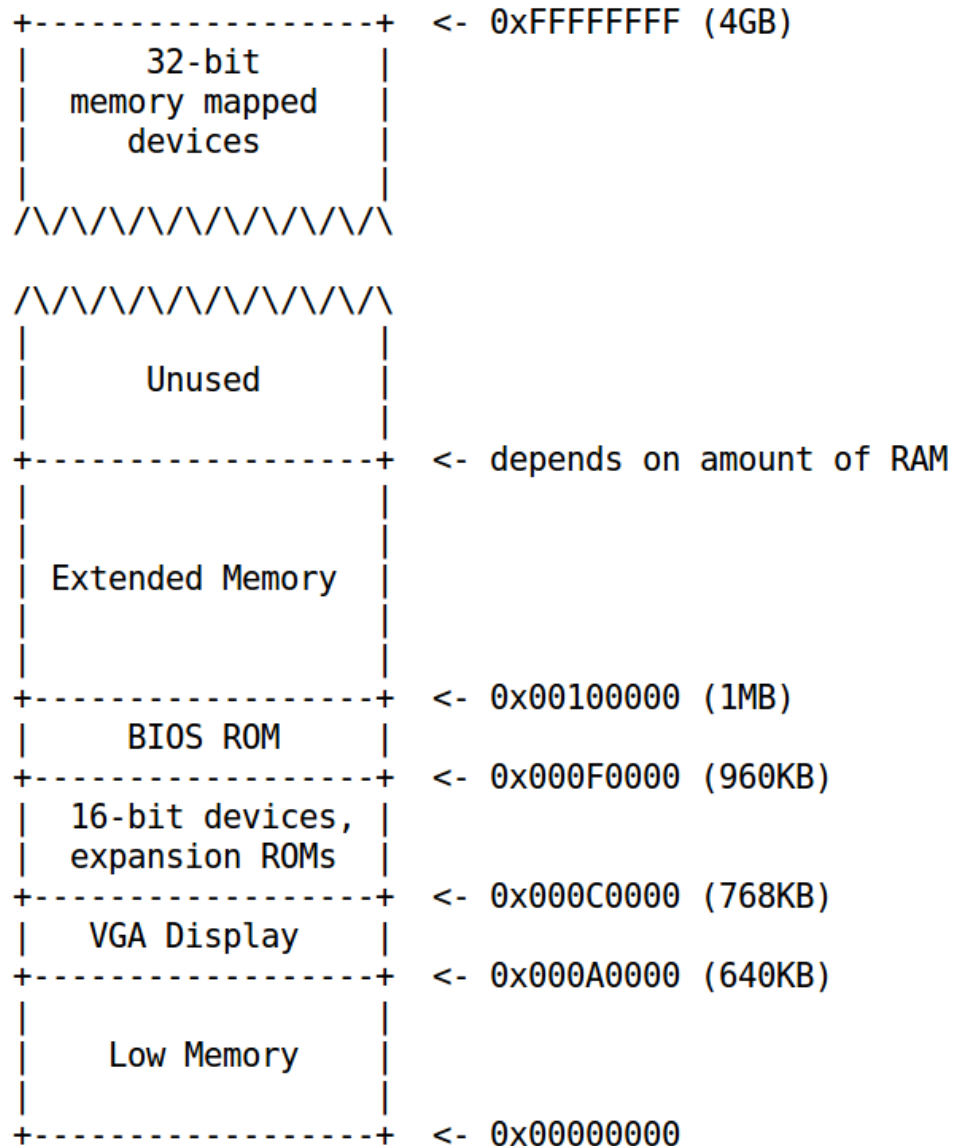
Inizio indirizzi: 0x00000000
Fine indirizzi: 0x000FFFFF

Il layout della memoria al momento del boot è questo

prima istruzione all'indirizzo:
=0xFFFF0
(convenzione)

SOLAB2 : MIT JOS lab 1

booting a PC



Spazio degli indirizzi fisici di un PC

16 pezzi da 2^{16} :

10 liberi
 2 per la VGA
 3 per altri device a 16 bit
 1 per il BIOS

SOLAB2 : MIT JOS lab 1

booting a PC

The ROM BIOS

Fate doppio click sulla finestra di QEMU

make qemu-nox-gdb

Questo comando avvia il kernel JOS e lo pone in attesa di una sessione gdb

Premete alt+F1 (per aprire un secondo terminale)
Portatevi nel medesimo folder in cui avete avviato JOS con il comando precedente

gdb

SOLAB2 : MIT JOS lab 1

booting a PC

The ROM BIOS - Exercise 2

Use GDB's **si** (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at Phil Storrs I/O Ports Description, as well as other materials on the 6.828 reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

SOLAB2 : MIT JOS lab 1

booting a PC

Comandi utili in gdb

<code>x/5i 0xfc867</code>	stampa 5 istruzioni assembly a partire da indirizzo (0xfc867)
<code>print /t 0x9fffffff</code>	stampa 0x9fffffff in binario
<code>info registers</code>	stampa informazioni sui registri
<code>print /d \$eax</code>	stampa contenuto eax in dec
<code>print /x \$eax</code>	" " hex
<code>print /t \$eax</code>	" " bin
<code>b *address</code>	imposta breakpoint all'indirizzo specificato
<code>b functionname</code>	imposta breakpoint alla funzione indicata
<code>c</code>	continua (eventualmente fino a prossimo breakpoint)
<code>si</code>	esegui una istruzione macchina

SOLAB2 : MIT JOS lab 1

booting a PC

The ROM BIOS - Exercise 2

La prima cosa che fa il BIOS è saltare all'indietro (è a soli 16 byte dalla fine della parte di memoria ad esso dedicata ...

Salta in basso a 0xF000:0xE05B

Poi:

```
jmp     0xfc85e
mov     %cr0, %eax
and     $0x9fffffff, %eax
mov     %eax, %cr0
cli
cld
```

provate in gdb: print /t 0x9ffffff

10011111111111111111111111111111

↑
Questi de bit di cr0 sono cache disable e not write-through

disabilita interrupts

clears the direction flag

SOLAB2 : MIT JOS lab 1

booting a PC

The ROM BIOS - Exercise 2

Il primo device ad essere toccato è NMI (non maskable interrupt)

```
mov    $0x8f, %eax
out    %al, $0x70      NMI enable
in     $0x71, %al     Real Time Clock
```

...

Imposta %ss=0 e %esp = 0x7000 per formare SS:[ESP]

...

Abilita A20 line (<http://www.win.tue.nl/~aeb/linux/kbd/A20.html>)

Poi lidt e gdtw (load interrupt descriptor table e global
Descriptor table)

SOLAB2 : MIT JOS lab 1

booting a PC

The ROM BIOS - Exercise 2

```
mov  %cr0, %eax  
or   $0x1, %eax  
mov  %eax, %cr0
```

Imposta il primo bit di cr0 (PE, protect enable) e poi esegue un far jump:

```
ljmp $0x8, $0xfc74c
```

SOLAB2 : MIT JOS lab 1

booting a PC

The ROM BIOS - Exercise 2

Lo schema generale di quello che succede è questo:

- all'avvio il PC **deve** iniziare ad eseguire nell'area di memoria contenente il BIOS (solo qui può trovare codice da eseguire)
- Immediatamente dopo l'avvio il BIOS salta all'indietro
- imposta una interrupt descriptor table
- Inizia ad impostare i device di cui è a conoscenza
- se trova un disco avviabile legge da esso il primo settore (che contiene il bootloader) e gli passa il controllo

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader

I dischi per PC sono divisi in settori da 512 byte. Un settore è l'unità minima che può essere letta/scritta. Se il disco è avviabile il primo settore è detto settore di avvio e in esso risiede il bootloader. Quando il BIOS trova un disco avviabile carica il settore di avvio (512 byte) in memoria all'indirizzo fisico **0x7c00**.

In seguito usa una istruzione `jmp` per impostare `CS:IP` a `0000:7c00` passando il controllo al bootloader.

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader ... cosa fa?

1. manda il processore da modalità reale in modalità protetta a 32 bit (poichè solo così il software può accedere alla memoria posta oltre il singolo MB di memoria che CPU può indirizzare. La traduzione degli indirizzi segmentati (segment:offset) in indirizzi fisici avviene in modo diverso in modalità protetta e che, dopo la transizione in modalità protetta gli offset sono a 32 bit e non più a 16 bit.
2. il bootloader legge il kernel dal disco e lo carica in memoria

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader ... i suoi sorgenti dove sono?

Il boot loader consiste di un sorgente assembly, **boot/boot.S**, e un sorgente C, **boot/main.c** .

Esaminate attentamente questi sorgenti e cercate di capire cosa fanno.

Dopo aver esaminato I sorgenti potete guardare anche il contenuto del file `obj/boot/boot.asm` che è il sorgente disassemblato dopo la sua compilazione da parte di `make`.
(utile per debug)

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader

Chiudete la sessione gdb di esercizio 2 se è ancora aperta e avviate una nuova. Impostate un breakpoint in corrispondenza dell'indirizzo a cui viene caricato il bootloader:

```
b *0x7c00
```

Usate *si* ed esaminate le istruzioni in memoria *x/i* . Cercate di rispondere alle seguenti domande (aiutatevi con il contenuto di `obj/boot/boot.asm`)

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader – Exercise 3

Set a breakpoint at address **0x7c00**, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `boot/boot.S`, using the source code and the disassembly file `obj/boot/boot.asm` to keep track of where you are. Also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.

Trace into `bootmain()` in `boot/main.c`, and then into `readsect()`. Identify the exact assembly instructions that correspond to each of the statements in `readsect()`. Trace through the rest of `readsect()` and back out into `bootmain()`, and **identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk.** Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

SOLAB2 : MIT JOS lab 1

booting a PC

Comandi utili in gdb

x/5i 0xfc867	stampa 5 istruzioni assembly a partire da indirizzo (0xfc867)
print /t 0x9fffffff	stampa 0x9fffffff in binario
info registers	stampa informazioni sui registri
print /d \$eax	stampa contenuto eax in dec
print /x \$eax	" " hex
print /t \$eax	" " bin
b *address	imposta breakpoint all'indirizzo specificato
b functionname	imposta breakpoint alla funzione indicata
c	continua (eventualmente fino a prossimo breakpoint)
si	esegui una istruzione macchina

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader – Exercise 3

Rispondete alle seguenti domande:

- a) A che punto il processore inizia ad eseguire codice a 32 bit? Cosa esattamente causa il passaggio dalla modalità a 16 alla modalità a 32 bit?
- b) Qual'e' l'ultima istruzione eseguita dal bootloader e qual'e' la prima istruzione eseguita dal kernel appena caricato?
- c) Dove si trova la prima istruzione del kernel?
- d) COme fa il boot loader a decidere quanti settori deve leggere per caricare l'intero kernel dal disco? DOve trova questa informazione?

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader – Exercise 3

a) A che punto il processore inizia ad eseguire codice a 32 bit? Cosa esattamente causa il passaggio dalla modalità a 16 alla modalità a 32 bit?

boot.S

```
lgdt  gdt_desc
movl  %cr0, %eax
orl   $CR0_PE_ON, %eax
movl  %eax, %cr0
```

```
lgdtl (%esi)
fs  jl  7c33 <protcseg+0x1>

and   %al, %al

or    $0x1, %ax

mov   %eax, %cr0
```

boot.asm

```
[ 0:7c1e] => 0x7c1e: lgdtw 0x7c64
0x00007c1e in ?? ()
(gdb) si
[ 0:7c23] => 0x7c23: mov  %cr0, %eax
0x00007c23 in ?? ()
(gdb) si
[ 0:7c26] => 0x7c26: or   $0x1, %eax
0x00007c26 in ?? ()
(gdb) si
[ 0:7c2a] => 0x7c2a: mov  %eax, %cr0
```

GDB

Immediatamente dopo aver impostato ad 1 il Primo bit di cr0 (PE) fa un `ljmp CS:IP` ed entra in modalità protetta.

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader – Exercise 3

a) A che punto il processore inizia ad eseguire codice a 32 bit? Cosa esattamente causa il passaggio dalla modalità a 16 alla modalità a 32 bit?

Dopo aver effettuato il salto troviamo, in `boot.asm`, un `.code32` . La prima istruzione posta dopo di esso è la prima istruzione eseguita in modalità 32 bit (set data segment selector).

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader – Exercise 3

b) Qual'e' l'ultima istruzione eseguita dal bootloader e qual'e' la prima istruzione eseguita dal kernel appena caricato?

L'ultima istruzione del bootloader si trova nel file main.c (folder: boot) alla riga 58:

```
// call the entry point from the ELF header
// note : does not return!
// ((void (*)(void)) (ELFHDR->e_entry))();
```

L'indirizzo di ingresso del kernel, che corrisponde alla prima istruzione che verrà eseguita dal kernel, è in e_entry.

Data questa informazione, riuscite ad identificare la prima istruzione eseguita dal kernel?

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader – Exercise 3

c) Dove si trova la prima istruzione del kernel?

Vari modi per rispondere ...

(GDB)

Cercare in `/obj/boot/boot.asm` indirizzo corrispondente a al punto in cui viene passato il controllo al kernel (`0x7d5e`)

Usare questa informazione per settare breakpoint in GDB e guardare la prima istruzione eseguita (si)

```
0x10000c      movw    $0x1234, 0x472
```

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader – Exercise 3

c) Dove si trova la prima istruzione del kernel?

Vari modi per rispondere ...

(objdump)

```
objdump -x obj/kern/kernel | less
```

Alla quinta riga dell'output troviamo:

start address 0x0010000c (il che è consistente con metodo precedente)

... il kernel si aspetta di essere eseguito all'indirizzo 0x0010000c

SOLAB2 : MIT JOS lab 1

booting a PC

The boot loader – Exercise 3

d) Come fa il boot loader a decidere quanti settori deve leggere per caricare l'intero kernel dal disco? Dove trova questa informazione?

La trova nell' header ELF.

ELFHDR->e_phnum

```
// load each program segment (ignores ph flags)
ph = struct( Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
for(; ph < eph; ph++)
```

Codice rilevante per la risposta: bootmain() presente in main.c
Durante caricamento in memoria dei vari segmenti ph → p_offset
conosce il punto iniziale in cui il blocco di kernel corrente va caricato e
la lunghezza del blocco : ph->p_memsz

SOLAB2 : MIT JOS lab 1

booting a PC

Loading the kernel – Exercise 5

Trace through the first few instructions of the **boot loader** again and identify the **first instruction** that would "break" or otherwise do the wrong thing if you were to get the boot loader's **link address** wrong. Then change the link address in boot/Makefrag to something wrong, run make clean, recompile the lab with make, and trace into the boot loader again to see what happens. Don't forget to change the link address back and make clean again afterward!

SOLAB2 : MIT JOS lab 1

booting a PC

Loading the kernel – Exercise 5

Un binario ELF inizia con un ELF-header di lunghezza fissa, seguito da un program header di lunghezza variabile contenente tutte le sezioni del programma che dovranno essere caricate. Definizioni di questi ELF header sono in `inc/elf.h` le sezioni del programma a cui siamo interessati sono:

<code>.text</code>	istruzioni eseguibili
<code>.rodata</code>	read-only data
<code>.data</code>	dati inizializzati del programma (es. <code>Int x=5</code>)

SOLAB2 : MIT JOS lab 1

booting a PC

Loading the kernel – Exercise 5

C richiede che variabili non inizializzate globali abbiano un valore iniziale 0. Quindi non c'è necessità di immagazzinare il contenuto di `.bss` in un binario ELF. Il linker registra solo l'indirizzo e la dimensione della sezione `.bss`. Saranno il loader (o il programma) a gestire questa sezione.

Esaminiamo la lista completa di nomi, dimensioni e link addresses presenti nell'eseguibile del kernel JOS:

```
objdump -h obj/kern/kernel
```

Troveremo diverse informazioni (attenzione a LMA e VMA di `.text`)

SOLAB2 : MIT JOS lab 1

booting a PC

Loading the kernel – Exercise 5

LMA: Load address (`ph` → `p_pa`) sezione viene caricata qui

VMA: Link address (l'indirizzo a cui si aspetta di essere eseguita, nello spazio virtuale del processo).

Ora torniamo alla domanda ... stiamo cercando “the first instruction that would break or otherwise do the wrong thing if you were to get the boot loader's link address wrong”.

Si può provare a cambiare questo indirizzo (modificando il valore posto dopo `-Ttext` in `boot/Makefrag`).

SOLAB2 : MIT JOS lab 1

booting a PC

Loading the kernel – Exercise 5

Si può provare a cambiare questo indirizzo (modificando il valore posto dopo -Ttext in boot/Makefrag).

Fate una copia di backup del file obj/kern/kernel.asm. Cambiate il valore nella riga del file Makefrag da -Ttext 0x7c00 a -Ttext 0x1337

In lab usate i comandi
make clean
make

Provate a cercare differenze tra i file disassemblati (diff). E provate ad eseguire il kernel. Dopo correggete il problema/ make clean/ make.

SOLAB2 : MIT JOS lab 1

booting a PC

Loading the kernel – Exercise 5

Ora proviamo con il file boot/boot.out :

```
objdump -h obj/boot/boot.out
```

A differenza del caso del kernel qui LMA e VMA di .text hanno lo stesso valore. Il kernel sta dicendo al bootloader di caricarlo in memoria ad un indirizzo basso (1 megabyte) ma si aspetta di essere eseguito da un indirizzo più alto.

L'entry point del kernel possiamo ottenerlo mediante :

```
objdump -f obj/kern/kernel (start address 0x0010000c)
```

SOLAB2 : MIT JOS lab 1

booting a PC

Loading the kernel – Exercise 6

We can examine memory using GDB's `x` command. The GDB manual has full details, but for now, it is enough to know that the command `x/Nx ADDR` prints `N` words of memory at `ADDR`. (Note that both 'x's in the command are lowercase.)

Warning: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in `xorw`, which stands for word, means 2 bytes).

Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

SOLAB2 : MIT JOS lab 1

booting a PC

Loading the kernel – Exercise 6

Uscire da sessione gdb e riavviare. Poi in gdb:

```
b *0x7c00
b *0x7d5e
c
x/8x 100000
c
x/8x 100000
```

Cosa osservate? Che spiegazione riuscite a dare?

booting a PC

LMA del JOS kernel

```
(gdb) b *0x7c00  
Breakpoint 1 at 0x7c00  
(gdb) b *0x7d5e  
Breakpoint 2 at 0x7d5e
```

```
(gdb) c  
Continuing.  
[ 0:7c00] => 0x7c00: cli
```

```
Breakpoint 1, 0x00007c00 in ?? ()
```

```
(gdb) x/8x 0x100000
```

0x100000:	0x00000000	0x00000000	0x00000000	0x00000000
0x100010:	0x00000000	0x00000000	0x00000000	0x00000000

```
(gdb) c  
Continuing.
```

```
The target architecture is assumed to be i386
```

```
=> 0x7d5e: call *0x10018
```

```
Breakpoint 2, 0x00007d5e in ?? ()
```

```
(gdb) x/8x 0x100000
```

0x100000:	0x1badb002	0x00000000	0xe4524ffe	0x7205c766
0x100010:	0x34000004	0x0000b812	0x220f0011	0xc0200fd8

SOLAB2 : MIT JOS lab 1

booting a PC

The kernel – Exercise 7

Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at **0x00100000** and at **0xf0100000**. Now, single step over that instruction using the `stepi` GDB command. Again, examine memory at `0x00100000` and at `0xf0100000`. Make sure you understand what just happened.

What is the first instruction after the **new mapping is established** that would fail to work properly if the mapping weren't in place? Comment out the `movl %eax, %cr0` in `kern/entry.S`, trace into it, and see if you were right.

SOLAB2 : MIT JOS lab 1

booting a PC

The kernel – Exercise 7

```
objdump -h obj/kern/kernel
```

Cosa rappresentano gli indirizzi 0x100000 e 0xf0100000? Sono, rispettivamente, il link address (VMA) al quale il kernel si aspetta di eseguire e l'indirizzo a cui il boot loader carica il kernel.

Fino al momento in cui kernentry.S imposta il flag CR0_PG i riferimenti alla memoria sono trattati come indirizzi fisici (ad essere precisi sono indirizzi lineari, ma boot/boot.S imposta un identity mapping da indirizzi lineari ad indirizzi fisici).

lineare = non paginato

SOLAB2 : MIT JOS lab 1

booting a PC

The kernel – Exercise 7

Dal momento in cui `CR0_PG` e' impostato I riferimenti ad indirizzi virtuali vengono tradotti dall'hardware di supporto alla memoria virtuale in indirizzi fisici.

A partire dal punto in cui eravamo alla fine dell'esercizio precedente:

Procediamo con pochi **si**

SOLAB2 : MIT JOS lab 1

booting a PC

The kernel – Exercise 7

Dal momento in cui `CR0_PG` e' impostato I riferimenti ad indirizzi virtuali vengono tradotti dall'hardware di supporto alla memoria virtuale in indirizzi fisici.

A partire dal punto in cui eravamo alla fine dell'esercizio precedente:

Procediamo con pochi `si` fino a quando non raggiungiamo l'istruzione `mov %eax, %cr0`

SOLAB2 : MIT JOS lab 1

booting a PC

The kernel – Exercise 7

Dopo aver raggiunto il punto richiesto dall'esercizio scriviamo:

```
x/10x 0x100000  
x/10x 0xf0100000  
si  
x/10x 0x100000  
x/10x 0xf0100000
```

Cosa osservate? E che spiegazione daresti?

SOLAB2 : MIT JOS lab 1

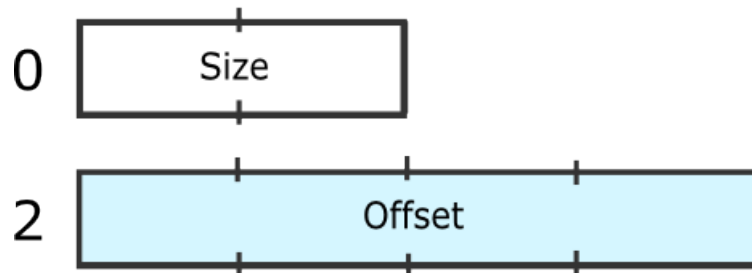
booting a PC

GDT (caricata da istr. assembly LGDT)

Global Descriptor Table. Fornisce a CPU informazioni su segmenti di memoria

Si aspetta come argomento un GDT Descriptor. Offset = indirizzo (lineare) di GDT stessa, Size = numero entries

GDT Descriptor



Ogni entry (64 bit) ha questa struttura

31				16				15				0							
Base 0:15								Limit 0:15											
63		56		55		52		51		48		47		40		39		32	
Base 24:31				Flags				Limit 16:19				Access Byte				Base 16:23			

Limit, valore a 20 bit composto da Significato: **massimo numero di Unità di indirizzamento (in termini di pagine o di byte)**

SOLAB2 : MIT JOS lab 1

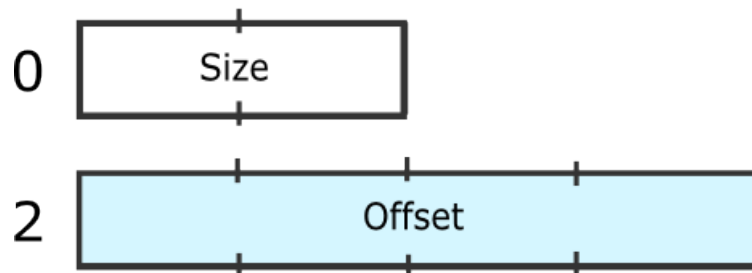
booting a PC

GDT (caricata da istr. assembly LGDT)

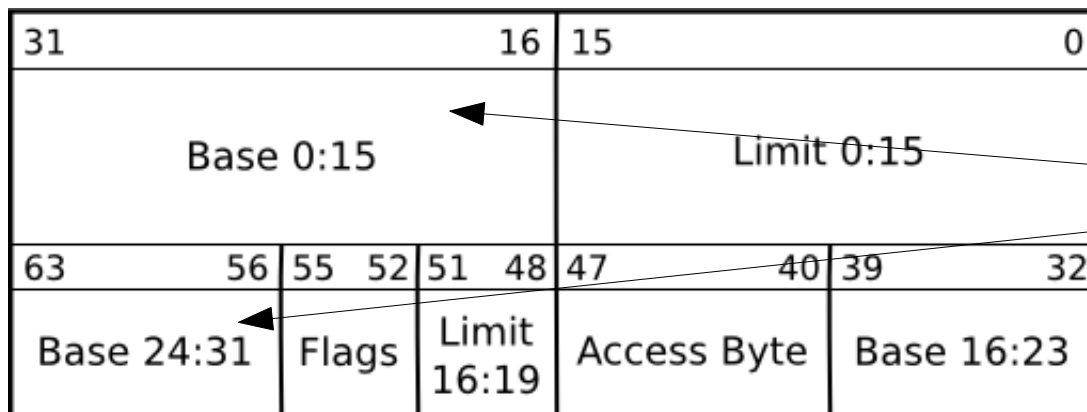
Global Descriptor Table. Fornisce a CPU informazioni su segmenti di memoria

Si aspetta come argomento un GDT Descriptor. Offset = indirizzo (lineare) di GDT stessa, Size = numero entries

GDT Descriptor



Ogni entry (64 bit) ha questa struttura



Base, valore a 32 bit composto da Significato: **indirizzo lineare a cui inizia il segmento di memoria**

SOLAB2 : MIT JOS lab 1

booting a PC

PARTE 1b (esercizi 8-12)

re@di.unimi.it

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Leggere kern/printf.c, lib/printfmt.c, e kern/console.c, dobbiamo cercare di capire le relazioni tra questi file. Dovremo capire come mai printfmt.c è posto in una directory separata (lib).

Exercise 8. We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

```
grep 'octal' kern/printf.c
grep 'octal' kern/console.c
grep 'octal' lib/printfmt.c
```

dimostrano che la parola octal è presente solo in lib/printfmt.c .
Iniziamo ad indagare da questo file.

SOLAB2 : MIT JOS lab 1

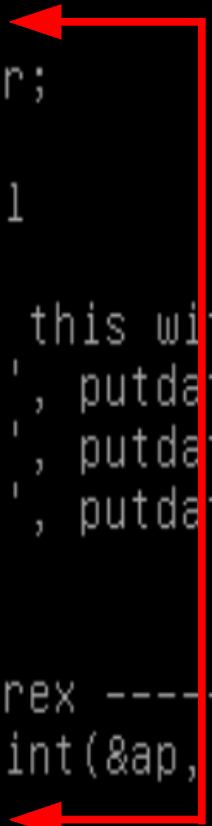
booting a PC

In void vprintfmt

```
// unsigned decimal
case 'u':
    num = getuint(&ap, lflag);
    base = 10;
    goto number;

// (unsigned) octal
case 'o':
    // Replace this with your code.
    //putch('X', putdat);
    //putch('X', putdat);
    //putch('X', putdat);
    //break;

//----- rex -----
num = getuint(&ap, lflag);
base = 8;
goto number;
//-----
```



SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

1) Explain the interface between `printf.c` e `console.c`

In `kern/printf.c` è definita la funzione **`putch`**. A sua volta `putch` chiama **`cputchar`** (in `kern/console.c`).

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

2) Explain the following from `kern/console.c`

```
1  if (crt_pos >= CRT_SIZE) {
2      int i;
3      memcpy(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
4      for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5          crt_buf[i] = 0x0700 | ' ';
6      crt_pos -= CRT_COLS;
7  }
```

Il testo dell'esercizio e il sorgente differiscono per l'utilizzo di `memcpy` e di `memmove`. Arrivati al fondo di una schermata è necessario copiare le righe, riscrivere lo schermo dalla seconda riga in poi, ed aggiungere una nuova riga di testo . Scrolling.

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

3) Trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;  
cprintf("x %d, y %x, z %d\n", x, y, z);
```

In the call to `cprintf()`, to what does `fmt` point? To what does `ap` point?

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

3) Trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;  
cprintf("x %d, y %x, z %d\n", x, y, z);
```

In kern/monitor.c PRIMA della prima chiamata a cprintf() inserisco:

```
//----- rex -----  
int x = 1, y = 3, z = 4;  
cprintf("x %d, y %x, z %d\n", x, y, z);  
//-----
```

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

3) Trace the execution of the following code step-by-step:

Torno nel root folder del laboratorio.

```
make clean
```

```
make qemu-gdb
```

```
alt+F2 (passo in un'altra shell ed effettuo il login
```

```
cd /home/jos/solab-jos
```

```
gdb
```

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

3) Trace the execution of the following code step-by-step:

In gdb

```
b cprintf
```

```
b vcprintf
```

```
c
```

```
bt 3
```

```
c
```

```
bt 3
```

```
c          finchè non raggiungiamo la chiamata che ci interessa
```

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

3) Trace the execution of the following code step-by-step:

Una volta raggiunta la chiamata che ci interessa (stampa x,y,z) :

```
bt 3
```

```
c
```

```
ct 3 ← qui vediamo l'indirizzo di ap
```

```
x/12x indirizzo_ap
```

ap è un array contenente i valori x,y,z (in quest'ordine)

fmt punta ad una stringa di formattazione

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

3) Trace the execution of the following code step-by-step:

Una volta raggiunta la chiamata che ci interessa (stampa x,y,z) :

```
bt 3
```

```
c
```

```
ct 3 ← qui vediamo l'indirizzo di ap
```

```
x/12x indirizzo_ap
```

ap è un array contenente i valori x,y,z (in quest'ordine)
fmt punta ad una stringa di formattazione. Uscire da gdb.

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

4) Run the following code.

He**110** World

```
unsigned int i = 0x00646c72;  
cprintf("H%x Wo%s", 57616, &i);
```

Char	Dec	Oct	Hex
d	100	0144	0x64
l	108	0154	0x6c
r	114	0162	0x72

What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

He**110** World

4) Run the following code.

```
unsigned int i = 0x00646c72;  
cprintf("H%x Wo%s", 57616, &i);
```

Char	Dec	Oct	Hex
d	100	0144	0x64
l	108	0154	0x6c
r	114	0162	0x72

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set i to in order to yield the same output? **0x726c6400**

Would you need to change 57616 to a different value? **No**

SOLAB2 : MIT JOS lab 1

booting a PC

Formatted printing to the console

Domande:

5) In the following code, what is going to be printed after 'y='?
(note: the answer is not a specific value.) Why does this happen?

```
cprintf("x=%d y=%d", 3);
```

Il valore di memoria successivo a dove è conservato il 3 (in ap)

SOLAB2 : MIT JOS lab 1

booting a PC

The stack

In the final exercise of this lab, we will explore in more detail the way the C language uses the stack on the x86, and in the process write a useful new kernel monitor function that prints a **backtrace of the stack**: a list of the saved Instruction Pointer (IP) values from the nested call instructions that led to the current point of execution.

Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

SOLAB2 : MIT JOS lab 1

booting a PC


The stack

Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel **reserve space** for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?


Lo stack è definito in kern/entry.S

per vedere come il kernel imposta lo stack andiamo in obj/kern/kernel.asm

```
# boot stack
#####
.p2align      PGSHIFT
.globl        bootstack
bootstack:
.space        KSTKSIZE
.globl        bootstacktop
```



```
# Set the stack pointer
movl    $(bootstacktop),%esp
f0100034:    bc 00 00 11 f0        mov    $0xf0110000,%esp
```



SOLAB2 : MIT JOS lab 1

booting a PC

The stack

Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel **reserve space** for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

Il puntatore allo stack è inizializzato per puntare all'indirizzo **bootstacktop** (indirizzo alto): le push lo faranno decrescere (lo spazio riservato disponibile termina a **bootstack**).

SOLAB2 : MIT JOS lab 1

booting a PC

The stack

Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the `test_backtrace` function in `obj/kern/kernel.asm`, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words?

Se è aperta una sessione gdb terminarla usando quit

In tty1 scrivere: `make qemu-gdb`

Spostarsi in tty2 e avviare gdb

SOLAB2 : MIT JOS lab 1

booting a PC

The stack

In gdb (tty2) (dopo aver trovato indirizzo test_backtrace in kernel.asm):

```
b * 0xf0100040
```

```
c
```

da qui in poi procedere con alcuni si (poi c)

Risposta Exercise 10:

1 word per EIP, 1 per EBP, 1 per EBX, 5 per altri dati. Totale: 8 word

(ricordare di terminare sessione di debug in tty2)

SOLAB2 : MIT JOS lab 1

booting a PC

The stack

Leggere testo lab a monte di questo esercizio. Contiene informazioni utili per la sua soluzione.

Exercise 11. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run `make grade` to see if its output conforms to what our grading script expects, and fix it if it doesn't. After you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

If you use `read_ebp()`, note that GCC may generate "optimized" code that calls `read_ebp()` before `mon_backtrace()`'s function prologue, which results in an incomplete stack trace (the stack frame of the most recent function call is missing). While we have tried to disable optimizations that cause this reordering, you may want to examine the assembly of `mon_backtrace()` and make sure the call to `read_ebp()` is happening after the function prologue.

SOLAB2 : MIT JOS lab 1

booting a PC

The stack

Exercise 11. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run `make grade` to see if its output conforms to what our grading script expects, and fix it if it doesn't. The backtrace function should display a listing of function call frames in the following format:

Stack backtrace:

```
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2
00000031
```

```
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28
00000061
```

...

SOLAB2 : MIT JOS lab 1

booting a PC

The stack

Exercise 12 (part I) . Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.

In `debuginfo_eip`, where do `__STAB_*` come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- look in the file `kern/kernel.ld` for `__STAB_*`
- run `i386-jos-elf-objdump -h obj/kern/kernel`
- run `i386-jos-elf-objdump -G obj/kern/kernel`
- run `i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c`, and look at `init.s`.
- see if the bootloader loads the symbol table in memory as part of loading the kernel binary

Complete the implementation of `debuginfo_eip` by inserting the call to `stab_binsearch` to find the line number for an address.

SOLAB2 : MIT JOS lab 1

booting a PC

The stack

Exercise 12 (part II) . Each line gives the file name and line within that file of the stack frame's eip, followed by the name of the function and the offset of the eip from the first instruction of the function (e.g., `monitor+106` means the return eip is 106 bytes past the beginning of `monitor`).

Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

Tip: `printf` format strings provide an easy, albeit obscure, way to print non-null-terminated strings like those in STABS tables. `printf("%.s", length, string)` prints at most length characters of string. Take a look at the `printf` man page to find out why this works.

You may find that some functions are missing from the backtrace. For example, you will probably see a call to `monitor()` but not to `runcmd()`. This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the `-O2` from `GNUMakefile`, the backtraces may make more sense (but your kernel will run more slowly).

SOLAB2 : MIT JOS lab 1

booting a PC

The stack

Exercise 11/12 :

Partiamo dal 12... inkern/kdebug.c , Immediatamente dopo Your code here:

```
stab_binsearch (stabs, &lline, &rline, N_SLINE, addr);  
info->eip_line = stabs[lline].n_desc;
```

Questo permetterà di ottenere informazioni sul punto (numero riga relativo al sorgente) della chiamata.

SOLAB2 : MIT JOS lab 1

booting a PC

The stack

Exercise 11/12 :

Ora l'11... è più complesso. Richiede diverse modifiche in kern/monitor.c

a) Se dobbiamo aggiungere un comando al monitor è necessario aggiungere un elemento (comando, descrizione e nome funzione da chiamare) all'array di struct Command che contiene i comandi del monitor:

```
static struct Command commands[] = {  
    {"help", "Display this list of commands", mon_help},  
    {"kerninfo", "Display information about the kernel", mon_kerninfo},  
    {"backtrace", "Display current calling stack", mon_backtrace},  
};
```

SOLAB2 : MIT JOS lab 1

booting a PC

The stack

Exercise 11/12 :

Ora l'11... è più complesso. Richiede diverse modifiche in kern/monitor.c

b) l'operato di `mon_backtrace` richiede di manipolare delle locazioni di memoria e dei displacement costanti. Tanto vale aggiungere qualcosa che renda l'accesso meno prolisso e più chiaro nel codice a valle. Aggiungere immediatamente prima della funzione `mon_backtrace` quanto segue:

```
#define FORMAT_LENGTH 80  
#define EBP(_v) ((uint32_t)_v)  
#define EIP(_ebp) ((uint32_t)*(_ebp+1))  
#define ARG(_v,_cnt) ((uint32_t)*(_v+((_cnt)+2)))
```

SOLAB2 : MIT JOS lab 1

booting a PC

c) aggiungere nella funzione `mon_backtrace` quanto segue (dopo 'Your code here'):

```
int32_t cnt = 0;
uint32_t *addr = 0;
char format[FORMAT_LENGTH] = { 0 };
char formatName[FORMAT_LENGTH] = { 0 };
struct Eipdebuginfo info;
strcpy (format, "  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n");
strcpy (formatName, "  %s:%d: %.*s+%d\n");
addr = (uint32_t *) read_ebp ();
```

2 spazi

10 spazi

```
  cprintf ("Stack backtrace\n");
  for (; NULL != addr; cnt++)
  {
    cprintf (format, EBP (addr), EIP (addr), ARG (addr, 0), ARG (addr, 1),
            ARG (addr, 2), ARG (addr, 3), ARG (addr, 4));
```

```
    debuginfo_eip (EIP (addr), &info);
    cprintf (formatName,
            info.eip_file,
            info.eip_line,
            info.eip_fn_namelen,
            info.eip_fn_name, EIP (addr) - info.eip_fn_addr);
    //Trace the linked list implemented by Stack.
    addr = (uint32_t *) * addr;
  }
```

**Il tutto PRIMA di
return 0;**

SOLAB2 : MIT JOS lab 1

booting a PC

Ora è il momento di vedere se lo script che valuta il lavoro svolto approva la soluzione ... Portiamoci in /home/user/jos/solab.jos e usiamo questo comando:

make grade

```
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 377 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/jos/solab-jos'
running JOS: (11.6s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
```

Il laboratorio 1 è a posto ...