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

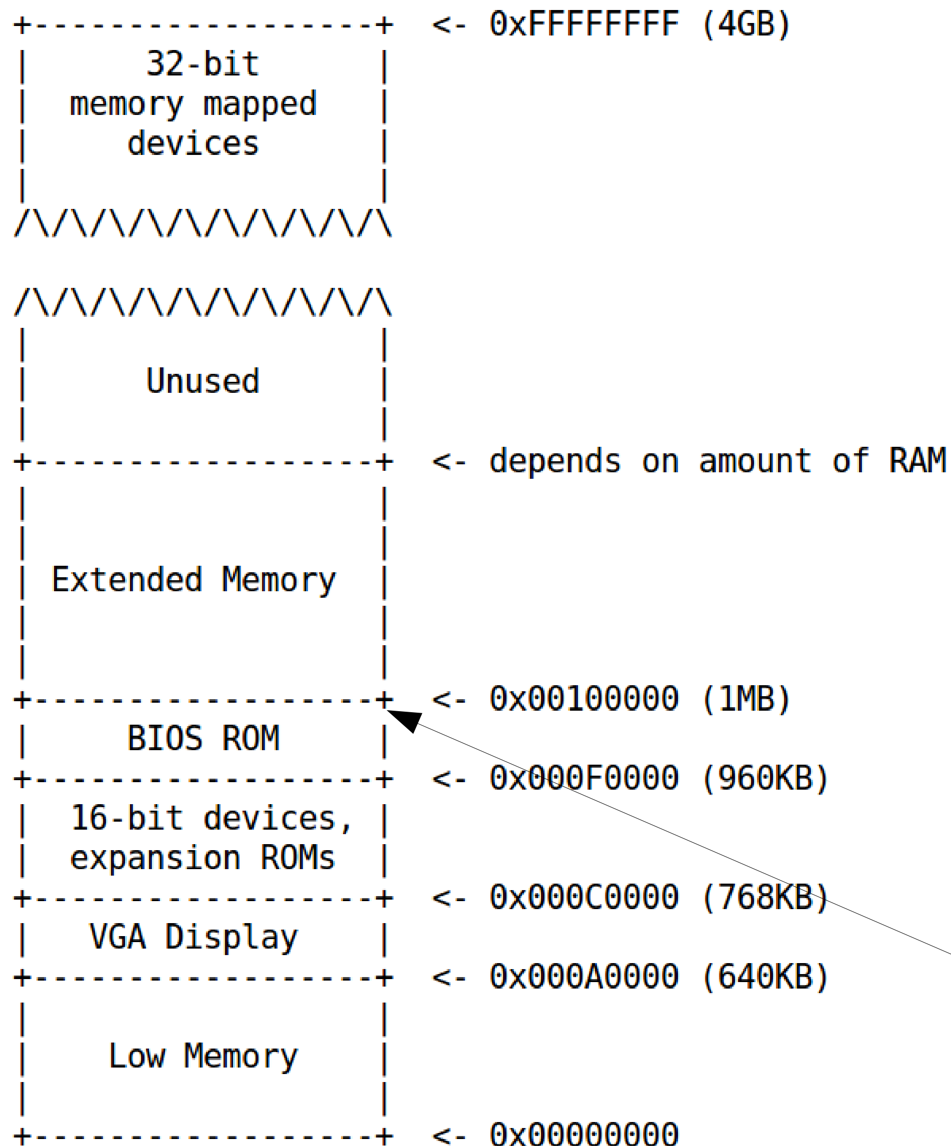
`L'ultimo comando utilizzato vi porta nel kernel monitor. Per uscire` **ctrl+C**

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

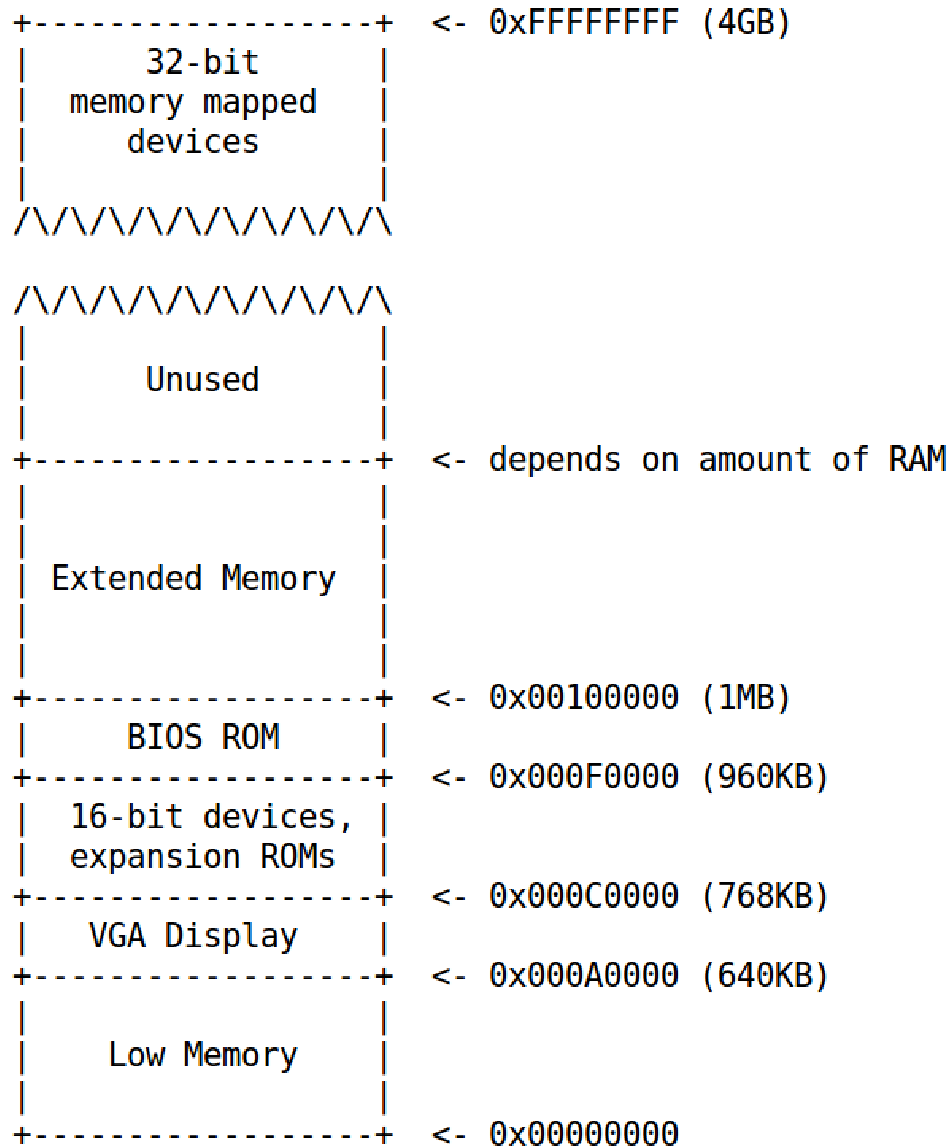
Processore **Intel 8088** può indirizzare 1 Mb di memoria fisica.

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

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 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

provate in gdb: print /t 0x9ffffff

Poi: 10011111111111111111111111111111

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

jmp	0xfc85e	
mov	%cr0, %eax	
and	$0x9fffffff, %eax	
mov	%eax, %cr0	
cli		disabilita interrupts
cld		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 lidtw 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 po esegue un far jump:

```
ljump $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 gi 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 avviavile 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 avviatene un'altra. Impostate un breakpoint in ciìorrispondenza 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    gdtdesc
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:

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

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 e 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 gbd e riavviare. Poi in gdb:

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

Cosa osservate? Che spiegazione riuscite a dare?

```
(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
```

LMA del JOS kernel

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
```

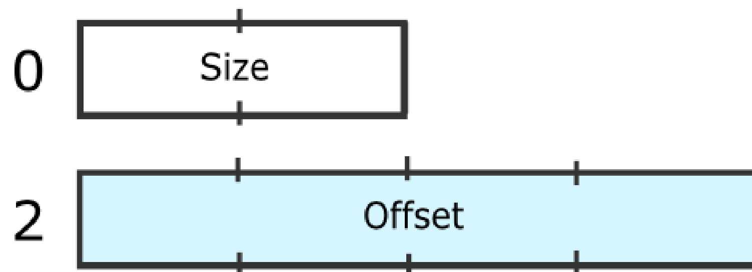
Cosa osservate? E che spiegazione dareste?

SOLAB2 : MIT JOS lab 1
booting a PC

GDT (caricata da istr. assembly LGDT)

Global Desctiptor 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



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

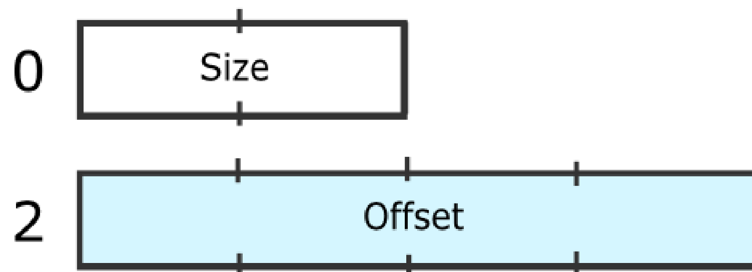
Significato: **massimo numero di Unità di indirizzamento (in termini di pagine o di byte)**

SOLAB2 : MIT JOS lab 1
booting a PC

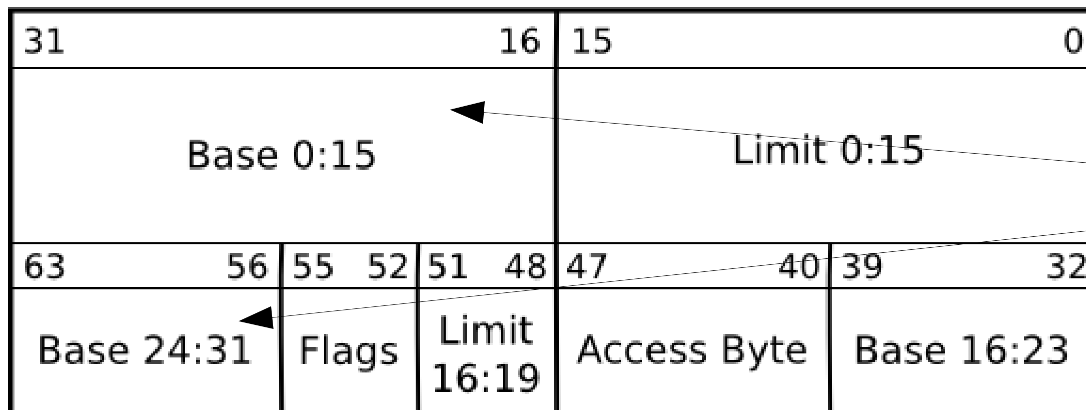
GDT (caricata da istr. assembly LGDT)

Global Desctiptor 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



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).

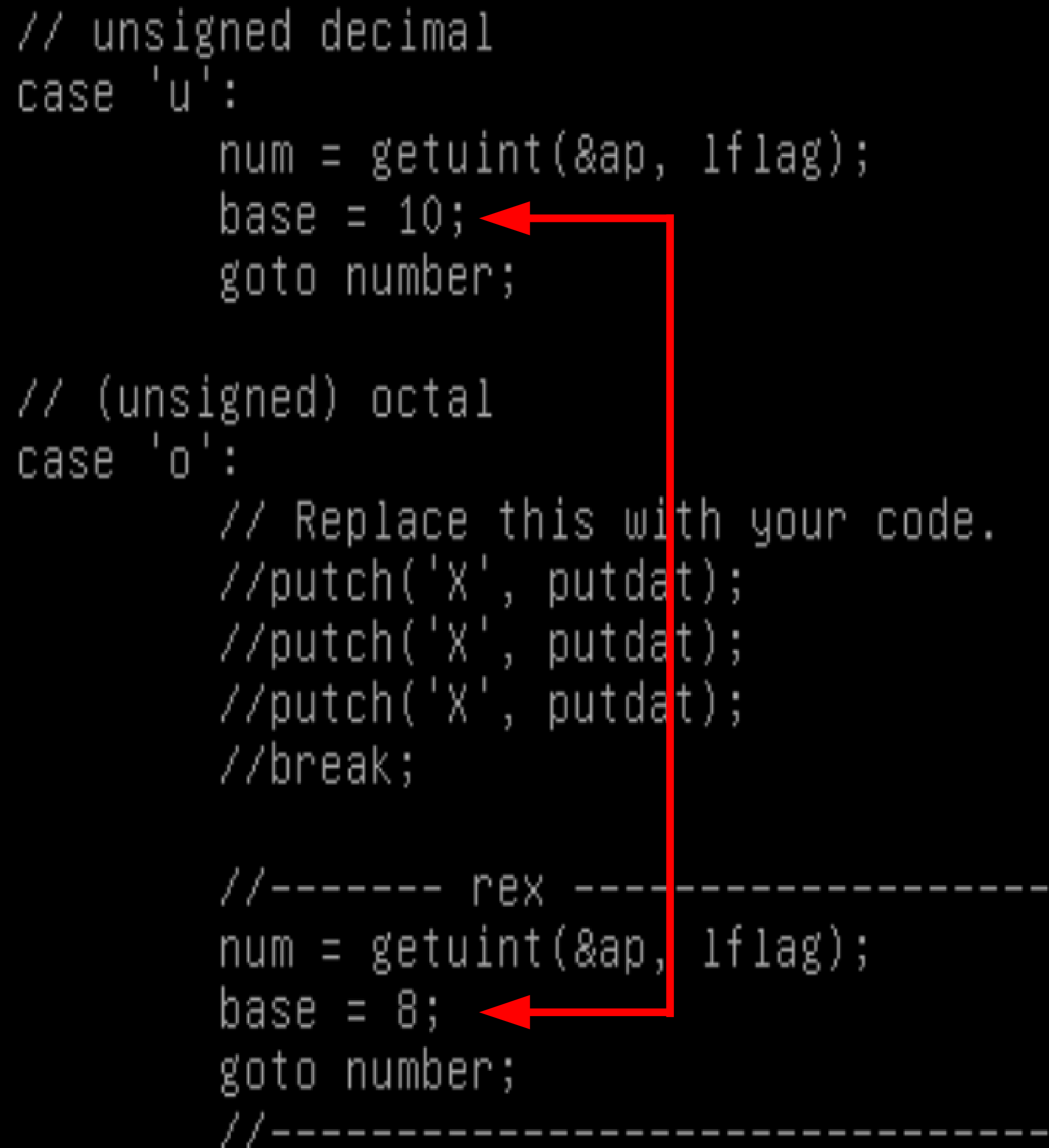
Exercise 8. We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

```
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.

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

He110 World

```
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:

He110 World

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

4) Run the following code.

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

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?

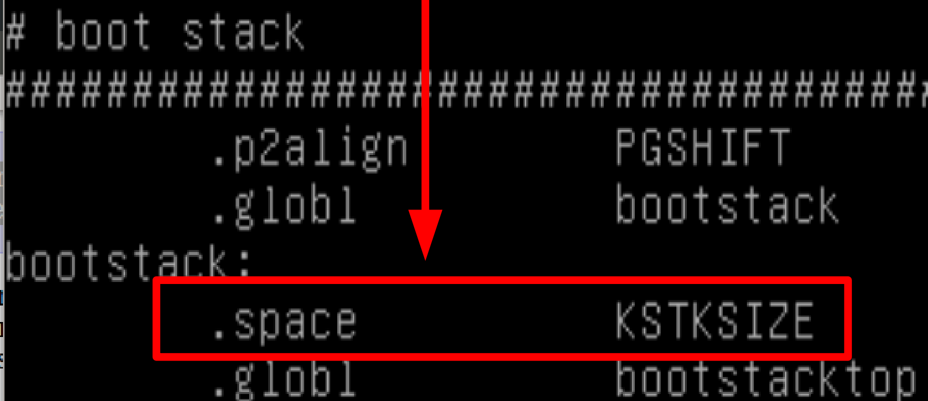
SOLAB2 : MIT JOS lab 1
booting a PC

The stack

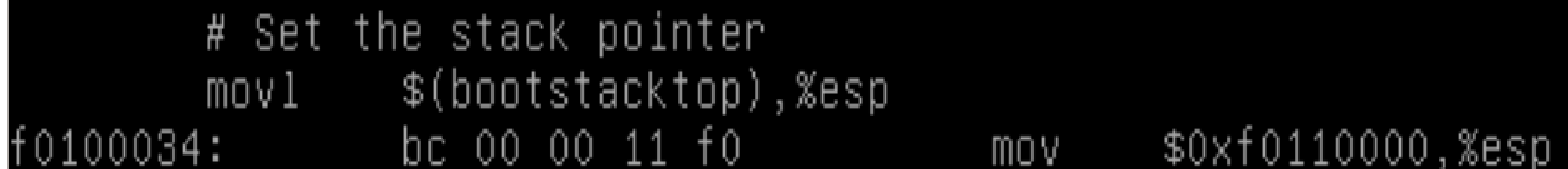
Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel **reserve space** for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

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 sessone 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 <u>aggiungere un comando</u> 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 ();

 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;
 }
```

2 spazi

10 spazi

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 ...

SOLAB2 : MIT JOS lab 1
booting a PC

Ricordiamoci di fare il commit della soluzione (dopo aver verificato che funziona) …

git commit -a -m 'Lab 1 solution'

SOLAB2 : MIT JOS lab 2
Memory management

Spostiamoci in /home/jos/solab-jos se non ci siamo già. Poi scarichiamo le soluzioni lab1 dal repository ufficiale solab:

git checkout lab1-2014

creare in /home/user il file .gdbinit contenente:
add-auto-load-safe-path /home/jos/solab-jos/.gdbinit

Impostiamo le varibili che definiscono l'dentità dell'utente:

git config --global user.email you@example.com
git config --global user.name “me”
git config --global core.editor “vi”

Facciamo il commit delle soluzioni scaricate:

git commit -a -m 'lab1-2014'

Ora scarichiamo I file aggiuntivi per Lab 2 (poi faremo il merge con le soluzioni):
git checkout -b lab2 origin/lab2
git show-branch
git merge lab1-2014

SOLAB2 : MIT JOS lab 2
Memory management

In questo laboratorio scriveremo il codice di gestione della memoria per il nostro sistema operativo. La gestione della memoria è realizzata da due componenti:

a) **Allocatore di memoria fisica per il kernel**. Permette al kernel di allocare/deallocare memoria. Questo componente opererà in unità di 4096 byte dette pagine. Il nostro obiettivo è quello di garantire la coerenza dello stato delle strutture dati che registrano quali pagine di memoria fisica sono libere e quali sono allocate, e quanti processi condividono ogni pagina allocata. Dovremo inoltre scrivere le routine per allocare e deallocare pagine di memoria.

b) **Memoria virtuale.** Questo componente mappa gli indirizzi virtuali utilizzati dal kernel e dal software che esegue in user space in indirizzi di memoria fisici. Tra le componenti hardware di x86 c'è MMU (memory management unit) che realizza il mapping quando una istruzione utilizza la memoria, consultando un insieme di tabelle (page tables). Modificheremo JOS in modo da creare e rendere utilizzabili le page tables necessarie alla MMU seguendo le specifiche fornite nei sorgenti sottoforma di commenti.

SOLAB2 : MIT JOS lab 2
Memory management

Lab2 contiene questi nuovi sorgenti:

inc/memlayout.h   Lab2 contiene questi nuovi sorgenti:
layout virtual address space da implementare modif. pmap.c

kern/pmap.c

kern/pmap.h
definisce struct PageInfo (serve x tener traccia di pg libere/usate)

kern/kclock.h

kern/kclock.c
interfaccia con CMOS RAM (bios salva qui quantità mem sistema)

SOLAB2 : MIT JOS lab 2
Memory management

Parte I : Physycal Page Management

Il sistema operativo deve tener traccia di di quali parti della RAM (mem fisica) sono correntemente in uso. JOS gestisce la memoria fisica in modo da poter utilzzare la MMU per mappare e proteggere in modo adeguato ogni porzione di memoria allocata.

Obiettivo di questa parte del laboratorio è scrivere il *physycal page allocator* (PPA). Esso tiene traccia di quali pagine sono libere utilizzando una lista concatenata di oggetti PageInfo, ognuno dei quali corrisponde ad una pagina di memoria fisica. Dobbiamo scrivere il PPA prima di scrivere le restanti parti del codice che implementano la memoria virtuale perché il codice che gestirà le page table **ha bisogno di memoria fisica** in cui allocare le page table.

SOLAB2 : MIT JOS lab 2
Memory management

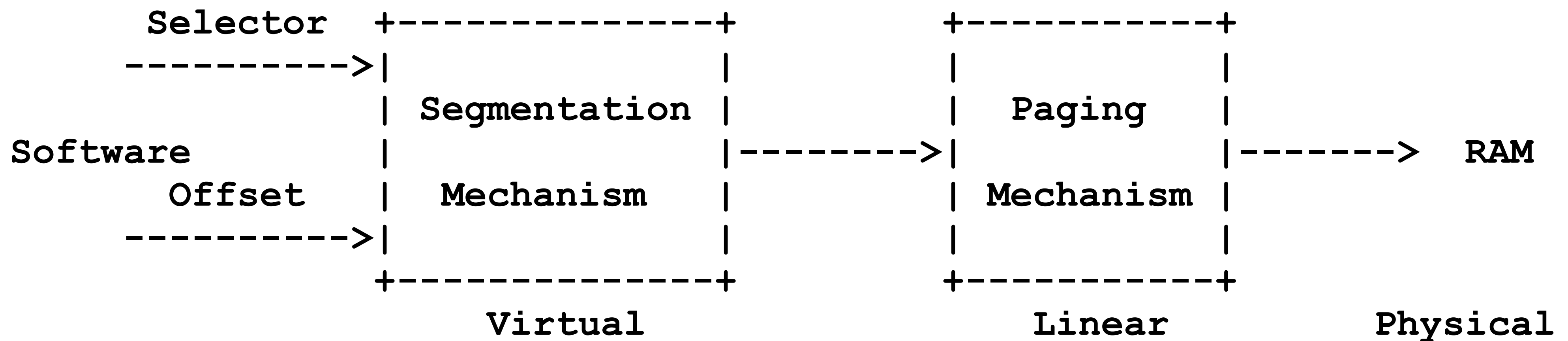
Exercise 1. In the file kern/pmap.c, you must implement code for the following functions (probably in the given order) :

boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()

check_page_free_list() and check_page_alloc() test your physical page allocator. You should boot JOS and see whether check_page_alloc() reports success. Fix your code so that it passes. You may find useful to add your own assert()s to verify that your assumptions are correct..

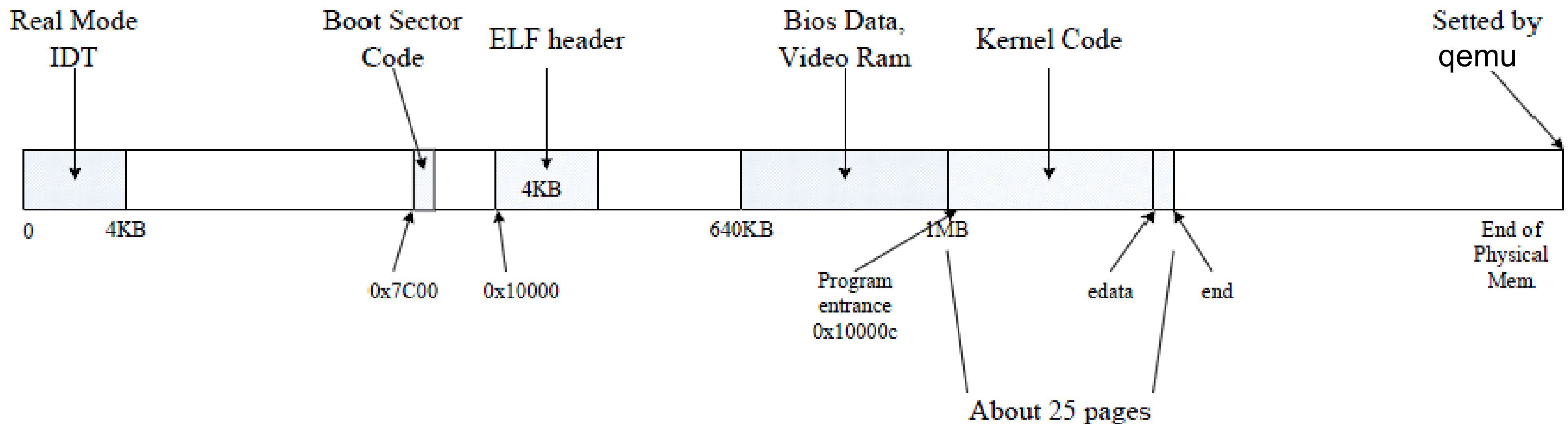
SOLAB2 : MIT JOS lab 2
Memory management

Questo laboratorio, come tutti I laboratori 6.828, richiede un piccolo sforzo investigativo per capire cosa dobbiamo fare. Infatti il materiale fornito per questo laboratorio non descrive tutti i dettagli del codice che è necessario aggiungere a JOS. E' di fondamentale importanza LEGGERE I SORGENTI DEI FILE DA MODIFICARE perché essi, spesso, contengono le specifiche del codice da realizzare e anche qualche suggerimento. E' anche necessario guardare i sorgenti delle altre parti di JOS, i manuali Intel e, magari, gli appunti di altri corsi che avete seguito in precedenza.



SOLAB2 : MIT JOS lab 2
Memory management

Schema di utilizzo della memoria fisica:



Possiamo osservare che, al momento, ci sono **4 aree di memoria fisica utilizzate**. La prima va da 0 a 4 KB in essa è presente IDT e alcune informazioni su di essa. All'indirizzo 0x7C00 troviamo il codice del bootloader. In corrispondenza di 0x10000 l'ELF header del kernel. Da 640 KB a 1MB inziano ad essere presenti routine e dati BIOS e IO. Da 0x100000 inizia il codice eseguibile del kernel. End è il punto in cui finisce l'area di memoria a disposizione del kernel. Notare che boot loader, ELF header dell'eseguibile del kernel e gli altri blocchi che abbiamo descritto **non sono più necessari** e possono essere sovrascritti. Purtroppo, al momento, non è possibile utilizzare la memoria fisica. Non possiamo richiedere aree di memoria e utilizzarle. Tutte queste (e altre) funzionalità dovranno essere implementate risolvendo Exercise 1.

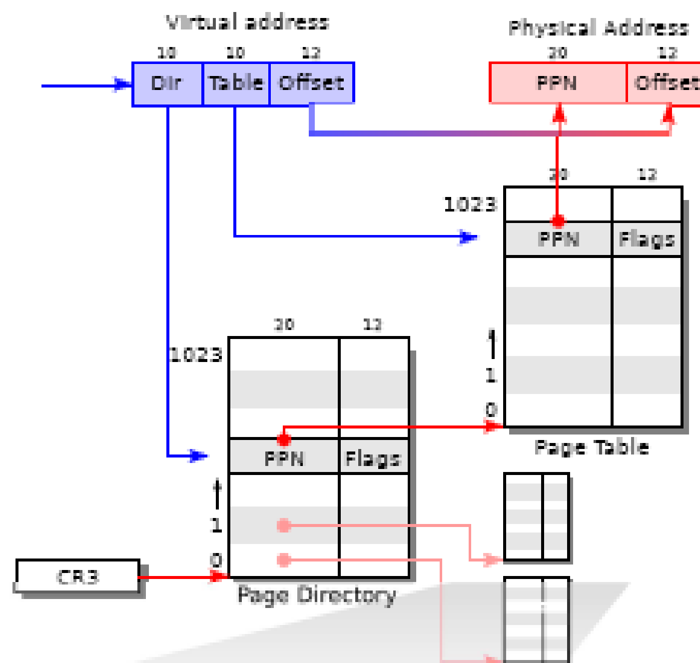
SOLAB2 : MIT JOS lab 2
Memory management

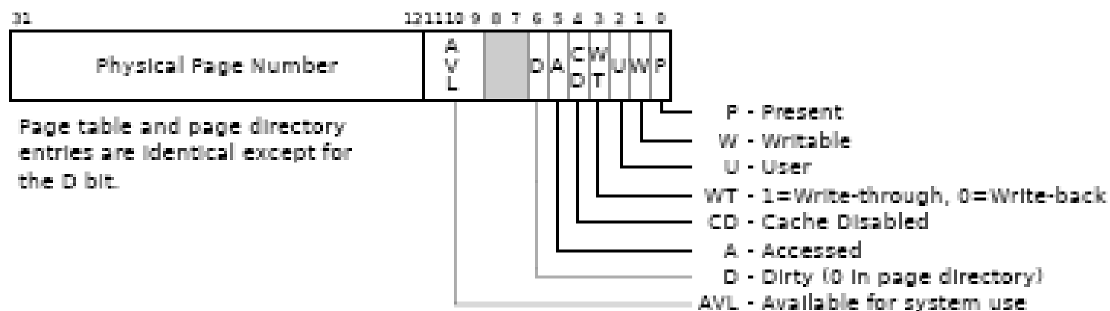
Schema di utilizzo della memoria fisica:

Le istruzioni x86 (sia user che kernel) vedono e manipolano unicamente indirizzi virtuali. La memoria fisica RAM è indicizzata utilizzando indirizzi fisici. Nell'architettura x86 esiste hardware che permette la conversione tra questi due tipi di indirizzi e che basa il suo funzionamento sull'utilizzo di page table. Una page tabel x86 è, dal punto di vista logico, un array do 2^20 page table entries (PTEs). Ogni **PTE** è composta da un physical page number (**PPN**) formato da 20 bit e alcuni flag che contengono informazioni/permessi. Lo hardware di paging traduce un indirizzo virtuale in un indirizzo fisico mappando i 20 bit alti dell'indirizzo virtuale in una page table (operazione che permette di identificare una PTE) e sostituendo questi 20 bit con il **PPN** contenuto nella **PTE**. Lo hardware di paging copia i 12 bit bassi dell'indirizzo virtuale in coda all'indirizzo fisico in corso di costruzione senza modificarli.
La page table, quindi, da al SO il controllo delle operazioni di traduzione da indirizzo virtuale a indirizzo fisico. Le quantità di memoria che possiamo allocare/deallocare **non** sono arbitrarie. Ogni operazione di allocazione/deallocazione coinvolge quantità di memoria aventi dimensione multiple di 2^12 bytes. Ognuno di questi blocchi di memoria di 4096 bytes sono detti **pagine**.

SOLAB2 : MIT JOS lab 2
Memory management

Il processo di traduzione da indirizzo virtuale a indirizzo fisico avviene in 2 step. Segue uno schema.





Il processo di traduzione avviene in 2 step La page table è presente in memoria sottoforma di albero a due livelli. La radice dell'albero è una **page directory** di 4096 byte che contiene 1024 riferimenti a *page table pages* . Ogni page table page è un **array di 1024 32 bit page table entries (PTEs)**.

Lo hardware di paging utilizza i **<u>10 bit alti</u>** di un indirizzo virtuale per selezionare un elemento della page directory. SE tale elemento è <u>presente</u> in page directory allora lo hardware di paging utilizza i **<u>10 bit seguenti</u>** nell'indirizzo virtuale per selezionare una page table entry (PTE) nella **page table page** puntata dalla entry selezionata nella page directory.

SOLAB2 : MIT JOS lab 2
Memory management

PTE_P indica se PTE è presente

PTE_W indica se è possibile scrivere in PTE (se non è impostato sono ammessi solo lettura e accesso)

PTE_U indica se programmi utente hanno il permesso di utilizzare la pagina (se non è impostato **SOLO IL KERNEL** ha accesso alla pagina)

Se la entry nella page directory **non esiste** o se non esiste nella page table page la PTE corrispondente al secondo blocco di 10 bit (a partire dall'alto) dell'indirizzo virtuale, lo hardware di paging lancia una eccezione.

Quale è il vantaggio di utilizzare questo approccio? La struttura a due livelli appena descritta permette di omettere intere **page table pages** nel caso, comune, in cui aree consistenti di memoria virtuale non abbiano un mapping in memoria fisica.

Ogni PTE contiene **diversi flag** che informa lo hardware di paging riguardo alle operazioni che è possibile effettuare sull'indirizzo virtuale associato all'indirizzo fisico.

Flags e strutture dati utilizzate dallo hardware di paging sono definiti in mmu.h

SOLAB2 : MIT JOS lab 2
Memory management

E1. boot.alloc():

boot_alloc() serve come mezzo temporaneo per allocare memoria. Ma perchè serve uno strumento da utilizzare solo inzialmente durante il processo di costruzione del sistema di gestione della memoria? Per costruire e attivare il sistema di gestione della memoria abbiamo la necessità di salvare strutture dati in memoria e, al momento, non ci sono strumenti per realizzare questo obiettivo. Serve un sistema che permetta di allocare e utilizzare memoria anche in queste fasi iniziali.

In assenza di un sistema di gestione delle pagine di memoria fisica la prima cosa da fare è costruire un **array di strutture** che rappresentano delle pagine ed associare ad ognuna di esse una quota di memoria. Una volta ottenuto questo risultato quando un programma richiederà della memoria sarà il S.O. ad assegnargli una quantità di pagine di memoria sufficiente a soddisfare le sue richieste (nel caso in cui la questa operazione sia possibile rispetto alla quantità di memoria disponibile nel sistema). Naturalmente il S.O. dovrà tenere traccia di quali pagine sono disponibili/utilizzate in un dato momento, e di altre caratteristiche specifiche di ogni singola pagina. Vedremo più avanti la funzione che alloca pagine fisiche per conto del S.O. Il suo nome è **page_alloc()**. Ma ci torneremo dopo.

SOLAB2 : MIT JOS lab 2
Memory management

E1. boot.alloc():

Alcune osservazioni su boot_alloc():

- il parametro **n** specifica la quantità (in **byte**) di memoria richiesta.
- se invochiamo boot_alloc passando **0 come argomento** otterremo la **prima posizione libera** nello spazio degli indirizzi attuale del kernel.

Osserviamo che la variabile **nextfree** è dichiarata come:

```
static char *nextfree; //virtual address of the next byte of free memory
```

quindi ogni chiamata a **boot_alloc()** farà riferimento allo stesso valore. nextfree punta all'indirizzo della prima posizione libera in memoria fisica. E' richiesto che nextfree sia allineata ad una dimensione costante da utilizzare per tutte le pagine di memoria (**PGSIZE**).

SOLAB2 : MIT JOS lab 2
Memory management

E1. boot.alloc():

Richiedere n byte di memoria, **se vogliamo mantenere l'allineamento di nextfree**, equivale a richiedere una porzione di memoria che parte dal valore corrente di nextfree e termina in nextfree + primo multiplo di PGSIZE in grado di contenere gli n byte. Per assicurarci di soddisfare questo vincolo possiamo utilizzare **ROUNDUP()**.
Notiamo inoltre la presenza di una variabile **end** dichiarata come segue:
extern char end[];

Basandoci sui commenti vediamo che end è un simbolo generato dal linker e che punta alla fine del segmento bss del kernel. Questo è il primo indirizzo virtuale che **non viene assegnato** al codice del kernel o a variabili globali. In definitiva è il primo indirizzo che possiamo utilizzare in sicurezza senza sovrascrivere accidentalmente qualcosa, e ,infatti, nextfree viene inizializzata a ROUNDUP(end, PGSIZE).

SOLAB2 : MIT JOS lab 2
Memory management

E1. boot.alloc():

Stando alle specifiche (commenti nel codice) se abbiamo esaurito la memoria fisica a disposizione boot_alloc() dovrebbe accorgersene e chiamare **panic()**. Tutto sta a capire <u>quando</u> siamo oltre il limite della memoria fisica disponibile. In pmap.c sono dichiarate due variabili:

```
size_t npages;
static size_t npages_basemem;
```

Esse sono impostate dalla funzione (sempre presente in pmap.c) **i386_detect_memory()**. In particolare può tornare utile npages. Essa corrisponde al <u>numero</u> di pagine di memoria fisica **disponibili**. Sarà quindi necessario utilizzare panic() quando la memoria richiesta è sopra al limite della quantità di memoria fisica disponibile:

`nextfree > KERNBASE + 0x00400000` (all'inizio sono mappati solo 4 MB)

SOLAB2 : MIT JOS lab 2
Memory management

E1. boot.alloc():

Inizialmente nextfree punta a end, fine della parte occupata dal kernel. Allocare n significa spostare il limite della regione occupata di n (arrotondando a PGSIZE).

```
// LAB 2: Your code here.

result = nextfree;
nextfree = ROUNDUP((char *) (nextfree + n), PGSIZE);
if ((uint32_t)nextfree > KERNBASE+0x00400000) { // out of memory
        panic("boot_alloc: Out of memory error\n");
}
return (void *)result;
```

SOLAB2 : MIT JOS lab 2
Memory management

E1. boot.alloc():

Chi chiama boot_alloc() e quando? Esistono due punti in cui boot_alloc() viene chiamata:

1. in mem_init() boot_alloc() viene chiamata per assegnare una pagina di memoria alla variabile **kern_pgdir** che verrà utilizzata come **page directory**.

2. sempre in mem_init() boot_alloc() verrà chiamata quando dovremo allocare una quantità di memoria corrispontente a npages volte la dimensione di una struct **PageInfo** salvando in **pages**. Il kernel utilizza questo array per tenere traccia delle pagine di memoria fisica e delle informazioni che le riguardano. Per ogni pagina di memoria fisica esiste una corrispondente struct PageInfo in questo array.

SOLAB2 : MIT JOS lab 2
Memory management

E1. mem_init():

mem_init() è la funzione principale che si occupa dell'inizializzazione del gestore della memoria che stiamo realizzando. Innanzitutto questa funzione chiama **i386_detect_memory()** la quale imposta il valore di **npages** (che verrà utilizzato da boot_alloc() come abbiamo visto prima). Prima di procedere alla modifica della funzione ricordiamoci di *commentare la chiamata a **panic()*** attiva quando la funzione non è stata ancora implementata. Gli obiettivi di mem_init() sono i seguenti:

i) creare una page directory per il kernel

ii) creare una struttura dati per la gestione delle pagine di memoria fisica.

iii) creare un sistema di gestione del mapping tra indirizzi virtuali e indirizzi fisici (questo comporta la creazione di una page table di "livello 2").

SOLAB2 : MIT JOS lab 2
Memory management

E1. mem_init():

In Exercise 1 quello che ci viene richiesto è di implementare le parti mancanti di questa funzione fino alla chiamata **check_page_free_list(1)**. Prima di procedere vediamo di ottenere informazioni più precise sulla struct PageInfo.

La struct è definita in **memlayout.h** e si nota che è costruita per poter essere utilizzata nella creazione di una lista concatenata (il nostro set di pagine di memoria fisica disponibili). Osservando attentamente la definizione di PageInfo troviamo anche delle informazioni utili riguardanti i campi al suo interno.

PageInfo Struct Reference

```
#include <memlayout.h>
```

Collaboration diagram for PageInfo:

[legend]

Data Fields

struct PageInfo *	pp_link
uint16_t	pp_ref

Detailed Description

Definition at line 175 of file memlayout.h.

SOLAB2 : MIT JOS lab 2
Memory management

E1. mem_init():

In Exercise 1 quello che ci viene richiesto è di implementare le parti mancanti di questa funzione fino alla chiamata **check_page_free_list(1)**. Prima di procedere vediamo di ottenere informazioni più precise sulla struct PageInfo.

La struct è definita in memlayout.h e si nota che è costruita per poter essere utilizzata nella creazione di una lista concatenata (il nostro set di pagine di memoria fisica disponibili). Osservando attentamente la definizione di PageInfo troviamo anche delle informazioni utili riguardanti i campi al suo inteno.

Esaminiamo la sua definizione in Inc/memlayout.h

PageInfo Struct Reference

`#include <memlayout.h>`

Collaboration diagram for PageInfo:

Data Fields

struct PageInfo *	pp_link
uint16_t	pp_ref

Detailed Description

Definition at line 175 of file memlayout.h.

SOLAB2 : MIT JOS lab 2
Memory management

E1. mem_init():

Ci sono molte informazioni utili nei commenti. Innanzitutto chiariscono un fatto importante: ogni struct PageInfo **non è** una pagina di memoria fisica ma serve per lo storage di **metadati riguardanti le pagine fisiche**.

Esiste una corrispondenza **1:1** tra pagine di memoria fisica e le struct PageInfo.

L'elemento **pp_link** all'interno della struct serve a concatenate le struct PageInfo tra di loro . Ogni elemento appartenente alla lista concatenata che andremo a creare rappresenta una pagina **LIBERA** in memoria fisica. **pp_ref** permette di risalire alla corrispondente pagina di memoria fisica.

Inoltre (e questa è una informazione utilissima!!!) è possibile ottenere **l'indirizzo in memoria fisica** della pagina corrispondente ad una particolare struct PageInfo utilizzando **page2pa()** che è definita in **kern/pmap.h** .

SOLAB2 : MIT JOS lab 2
Memory management

E1. mem_init():

page2pa() è definita così in kern/pmap.h .

```
00065 static inline physaddr_t
00066 page2pa(struct PageInfo *pp)
00067 {
00068         return (pp - pages) << PGSHIFT;
00069 }
```

Inoltre troviamo anche una seconda funzione che permette di effettuare il medesimo mapping <u>ma nella direzione opposta</u> (da indirizzo fisico a PageInfo).

```
00071 static inline struct PageInfo*
00072 pa2page(physaddr_t pa)
00073 {
00074         if (PGNUM(pa) >= npages)
00075                 panic("pa2page called with invalid pa");
00076         return &pages[PGNUM(pa)];
00077 }
```

SOLAB2 : MIT JOS lab 2
Memory management

E1. mem_init():

Ora procediamo con l’implementazione della parte di mem_init() richiesta in Exercise 1. La consegna, in conformità con i commenti presenti nel codice sorgente, è la seguente:

```
147          // Allocate an array of npages 'struct PageInfo's and store it in
'pages'.
00148        // The kernel uses this array to keep track of physical pages: for
00149        // each physical page, there is a corresponding struct PageInfo in
this
00150        // array.  'npages' is the number of physical pages in memory.
00151        // Your code goes here:
```

SOLAB2 : MIT JOS lab 2
Memory management

E1. mem_init():

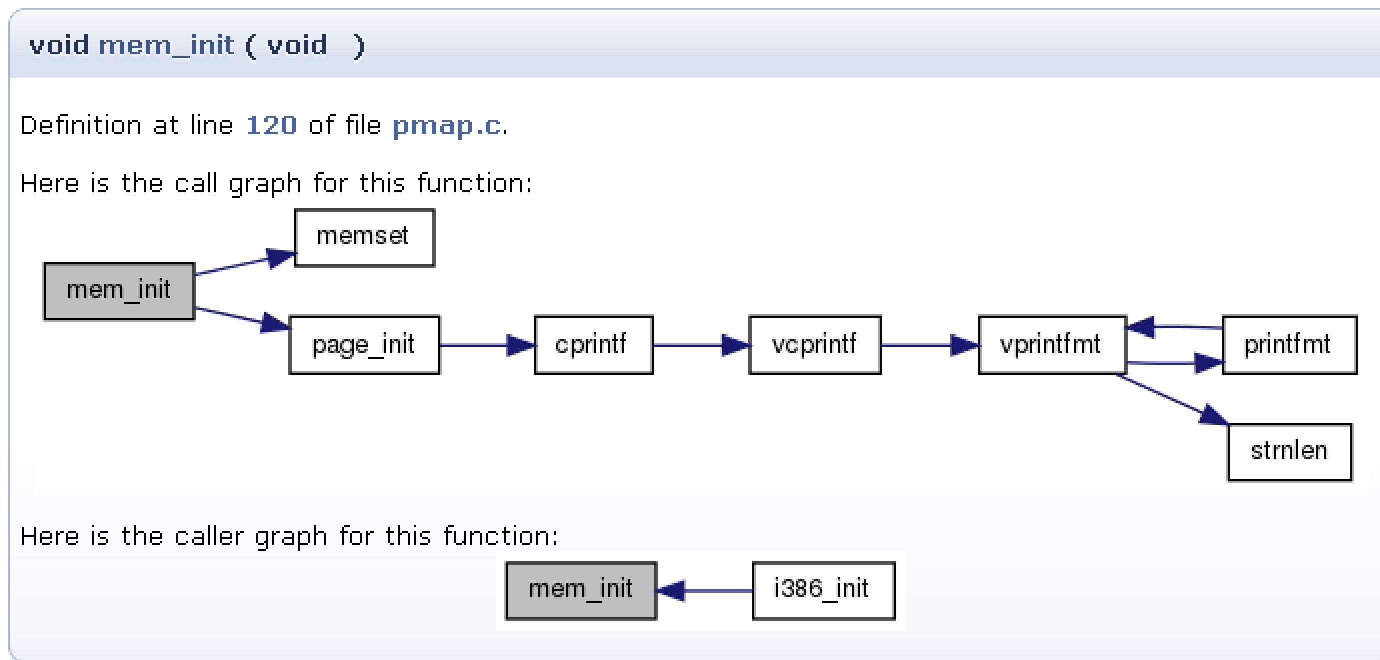
Ora procediamo con l'implementazione della parte di mem_init() richiesta in Exercise 1.

```
//////////////////////////////////////////////////////////////////////
// Allocate an array of npages 'struct PageInfo's and store it in '
// The kernel uses this array to keep track of physical pages: for
// each physical page, there is a corresponding struct PageInfo in
// array.  'npages' is the number of physical pages in memory.  Use
// to initialize all fields of each struct PageInfo to 0.
// Your code goes here:
n = sizeof(struct PageInfo) * npages;
pages = boot_alloc(0);
boot_alloc(n);
```

SOLAB2 : MIT JOS lab 2
Memory management

E1. mem_init():

Prima di lasciare mem_init cerchiamo di capire chi chiama (o è chiamato) dalla funzione. Grafici generati automaticamente, alcune chiamate mancano, ma sono utili.



SOLAB2 : MIT JOS lab 2
Memory management

E1. page_init():

Qui dobbiamo realizzare la seconda parte del sistema di gestione della memoria. Dobbiamo **inizializzare** le pagine libere nella lista concatenata **pages** (cfr. E1. mem_init()) associando a ciascuna di esse delle aree libere della memoria fisica.

pages è una lista concatenata di struct PageInfo. Il nostro obiettivo è quello di attraversare l'intera memoria fisica disponibile (pagina per pagina!) ed assegnare agli elementi di pages tutte le aree di memoria che **non sono utilizzate** (alcune sono utilizzate dal SO) e si trovano in regioni che non vengono mai mappate (tipo IO hole).

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_init():

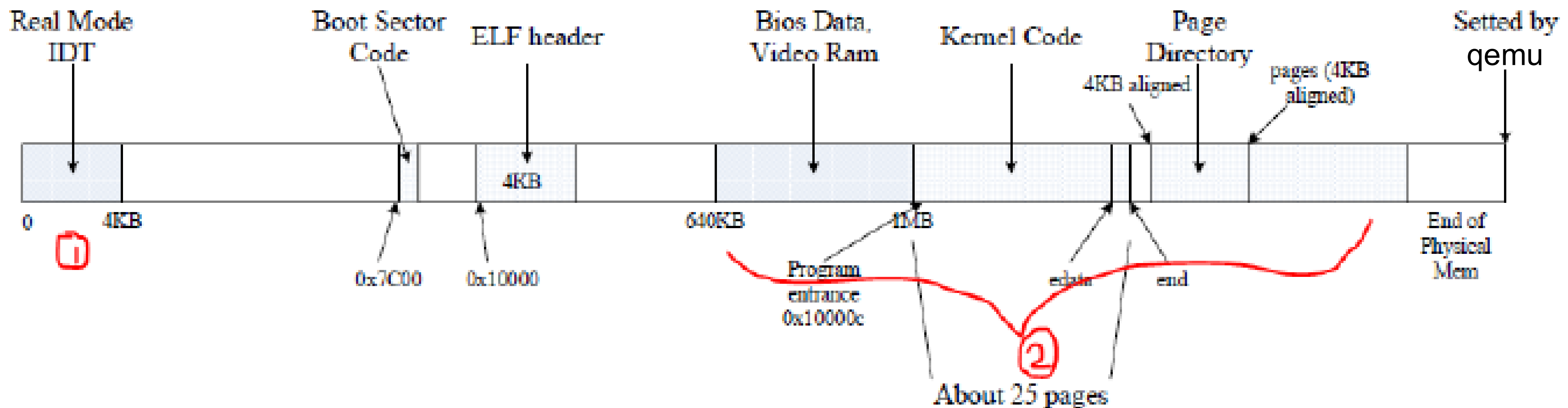
Per ottenere questo obiettivo dobbiamo conoscere qualche particolare in più rispetto allo stato attuale della memoria in modo da aggiungere alla lista pages (pagine di memoria fisica libera) solamente le parti di memoria che decideremo di marcare come libere (rendendole di fatto disponibili al S.O. in caso di necessità).

Lo schema attuale della memoria è riportato nell'immagine nella slide che segue. Notiamo che esistono aree occupate da elementi (es una contenente il codice caricato dal boot sector ed una seconda contenente l'ELF header del kernel) che **non sono più necessarie**. Quando le incontreremo potremo inserirle nella lista pages in modo da renderle **disponibili** al sistema. Le pagine **non inserite** nella lista vengono viste dal S.O. come in uso.

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_init():

Stato della memoria



SOLAB2 : MIT JOS lab 2
Memory management

E1. page_init():

Le specifiche (stando ai commenti nei sorgenti) sono queste:

```
00246 // Initialize page structure and memory free list.
00247 // After this is done, NEVER use boot_alloc again.  ONLY use the page
00248 // allocator functions below to allocate and deallocate physical
00249 // memory via the page_free_list.
00250 //
```

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_init():

Sono presenti anche altri suggerimenti utili:

```
00251 void
00252 page_init(void)
00253 {
00254         // The example code here marks all physical pages as free.
00255         // However this is not truly the case.  What memory is free?
00256         //  1) Mark physical page 0 as in use.
00257         //     This way we preserve the real-mode IDT and BIOS structures
00258         //     in case we ever need them.  (Currently we don't, but...)
00259
00260         //  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
00261         //     is free.
00262         //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
00263         //     never be allocated.
00264         //  4) Then extended memory [EXTPHYSMEM, ...).
00265         //     Some of it is in use, some is free. Where is the kernel
00266         //     in physical memory?  Which pages are already in use for
00267         //     page tables and other data structures?
00268         //
00269         // Change the code to reflect this.
00270         // NB: DO NOT actually touch the physical memory corresponding to
00271         // free pages!
```

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_init():

Tornando al problema da risolvere. Ci chiedono di considerare **libere due grandi aree di memoria**. La prima va dalla **seconda** pagina (dobbiamo preservare la prima perché avremo bisogno del suo contenuto in Lab3) fino a IOPHYSMEM (definita in memlayout.h, riga 92 valore 0x0A0000, 640 KB). La seconda parte che possiamo utilizzare è meno definita (nei suggerimenti almeno). Di certo non vogliamo sovrascrivere le strutture dati che abbiamo allocato utilizzando boot_alloc(). La posizione del primo byte di memoria libera è disponibile in **nextfree**. Quindi mi sembra prudente, come scelta, quella di considerare memoria libera tutte le aree:

al di sopra di PGSIZE (dimensione pagina 0)
non appartenenti all'intervallo che va da IOPHYSMEM a EXTPHYSMEM

Naturalmente dobbiamo ricordare l'iteratore che useremo per attraversare la lista si sposta di **una pagina alla volta**. Quindi dovremo dividere tutto per PGSIZE. In defintiva: ci sono **due aree di memoria che non vogliamo rendere disponibili e due aree di memoria che possiamo utilizzare**.

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_init():

Date queste premesse il tutto può essere risolto come segue ma prima di procedere notiamo che **i commenti ed il codice di esempio** mostrano come rendere libere le pagina (ossia come aggiungerle alla lista pages). Dovremo solo gestire l'altro caso in modo da **non** rendere disponibili (non inserire nella lista delle pagine libere) le pagine corrispondenti alle aree di memoria da salvaguardare.

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_init():

```
// free pages.

size_t i;
pages[0].pp_ref = 0;
pages[0].pp_link = NULL;
for (i = 1; i < npages_basemem; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
}
for (i = npages_basemem; i < EXTPHYSMEM/PGSIZE; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = NULL;
}
for (i = EXTPHYSMEM/PGSIZE; i < npages; i++) {
        pages[i].pp_ref = 0;
        if (page2kva(&pages[i]) < boot_alloc(0))
                pages[i].pp_link = NULL;
        else {
                pages[i].pp_link = page_free_list;
                page_free_list = &pages[i];
        }
}
```

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_init():

E anche questa è a posto. Nota utile: l’indirizzo del primo elemento della lista è contenuto in **page_free_list**. Potremo utilizzarlo in seguito per allocare / deallocare elementi da questa lista di pagine.

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_alloc():

Questa funzione serve per allocare una pagina di memoria. Svolge il ruolo che era svolto da boot_alloc() prima della creazione della lista di pagine di memoria disponibili (pages). D'ora in poi non dovremo più utilizzare boot_alloc() ma page_alloc() (e la corrispondente funzione per rendere nuovamente disponibili pagine utilizzate in precedenza, ossia page_free()). Allocare una pagina significa **rimuoverla dalla lista pages**. Queste sono le specifiche fornite nei commenti dei sorgenti:

```
00303 // Allocates a physical page.  If (alloc_flags & ALLOC_ZERO), fills the entire
00304 // returned physical page with '\0' bytes.  Does NOT increment the reference
00305 // count of the page - the caller must do these if necessary (either explicitly
00306 // or via page_insert).
00307 //
00308 // Returns NULL if out of free memory.
00309 //
00310 // Hint: use page2kva and memset
```

La funzione è relativamente semplice e quindi non mi dilungherò troppo su di essa. Quando S.O. chiede di allocare una pagina si rimuove la **prima pagina** dalla lista (useremo page_free_list).

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_alloc():

In conformità con le specifiche useremo **page2kva()** (che traduce un indice di pagina in un indirizzo di memoria virtuale **nello spazio di indirizzamento del kernel**) e **memset**. Inoltre dovremo gestire il caso in cui passiamo i parametri per indicare che vogliamo una pagina **riempita di '\0'**. E' importante notare che, per utilizzare memset, dobbiamo passare un **indirizzo virtuale**. E' a questo punto che tornerà utile l'utilizzo di page2kva() .

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_alloc():

```
struct PageInfo *
page_alloc(int alloc_flags)
{
	 // Fill this function in
	struct PageInfo * pp = page_free_list;
	if (!page_free_list) // out of memory
		return NULL;
	page_free_list = pp->pp_link;
	pp->pp_link = NULL;
	if (alloc_flags & ALLOC_ZERO){
		memset(page2kva(pp), 0, PGSIZE);
	}
	return pp;
}
```

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_free():

Anche questa funzione è abbastanza semplice. In conformità all'allocatore di memoria fisica che stiamo scrivendo rendere libera (e disponibile) una pagina significa **aggiungerla** alla lista delle pagine disponibili. Le specifiche fornite nei commenti sono queste:

```
00330 // Return a page to the free list.
00331 // (This function should only be called when pp->pp_ref
reaches 0.)
00332 //
```

L'unico punto a cui prestare attenzione è che non dovrebbe essere possibile liberare una pagina di memoria se esistono ancora dei riferimenti ad essa. Questo implica che, altrove, deve esistere un sistema che si occupa di incrementare/decrementare il numero di riferimenti attivi alla pagina in questione. Se e solo se il contatore di tali riferimenti vale 0 page_free() potrà rendere nuovamente disponibile la pagina.

SOLAB2 : MIT JOS lab 2
Memory management

E1. page_free():

```
void
page_free(struct PageInfo *pp)
{
        // Fill this function in
        // Hint: You may want to panic if pp->pp_ref is nonzero or
        // pp->pp_link is not NULL.
        assert(pp->pp_ref == 0);
        assert(pp->pp_link == NULL);
        pp->pp_link = page_free_list;
        page_free_list = pp;

}
```

Questo completa esercizio 1.

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Prima di fare qualsiasi altra cosa dobbiamo familiarizzare con i meccanismi propri dell’architettura x86 di gestione della memoria in modalità protetta.

Exercise 2

Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses paging for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it.

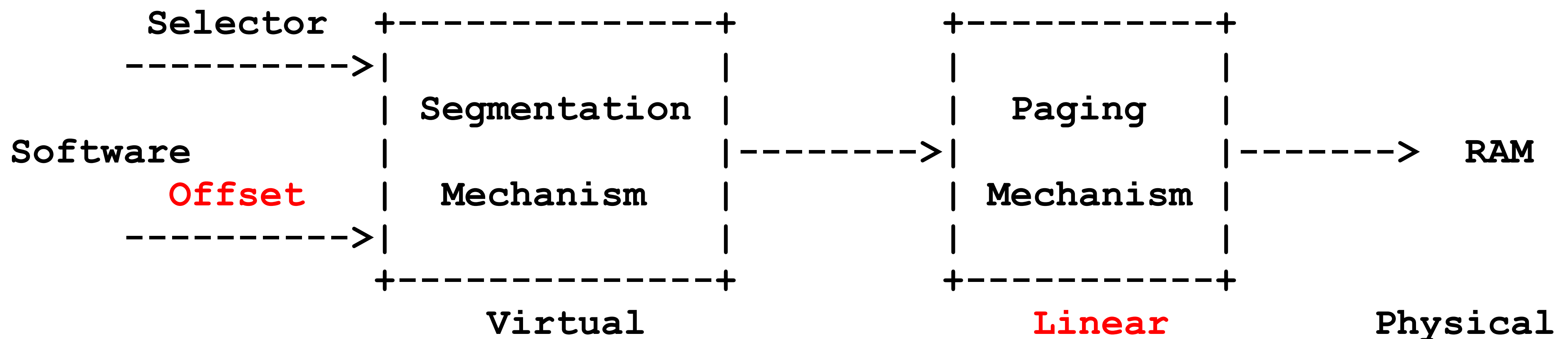
Soluzione: leggere il materiale indicato.

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Indirizzi lineari, virtuali e fisici:

Un indirizzo virtuale consiste di un segment selector e di un offset all'interno di un segmento. Un indirizzo lineare è quello che otteniamo dopo dopo la traduzione dell'indirizzo virtuale ma prima che il meccanismo di paginazione traduca l'indirizzo in un indirizzo fisico. Lo schema che riporto qui è già presente all'inizio del documento ma lo riporto per averlo a disposizione in questa sezione delle slide.



SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Indirizzi lineari, virtuali e fisici:

Un puntatore C è la *componente OFFSET dell'indirizzo virtuale*. In boot/boot.S è stata installata una Global Descriptor Table (GDT) che **disabilita il meccanismo di segment translation** semplicemente ponendo il base address di tutti i segmenti a 0 e impostando il limite a 0xffffffff . In questo modo il selettore (cfr. schema sopra) **non ha più effetto** e l'indirizzo lineare coincide sempre con l'**offset** dell'indirizzo virtuale. Al momento il meccanismo di segmentazione può essere tranquillamente ignorato. Esso diventerà importante in Lab3.

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Indirizzi lineari, virtuali e fisici:

Nella parte 3 di Lab1 è stata installata una semplice page table per permettere al kernel di eseguire in corrispondenza del suo link address (0xf0100000) anche se, al momento, il kernel si trova in memoria all'indirizzo 0x00100000. Questa page table mappa **solo 4 MB di memoria**. Nel layout della memoria virtuale che stiamo realizzando per JOS in questo laboratorio espanderemo questo mapping iniziale in modo da arrivare a mappare i primi **256 MB di memoria fisica** partendo dall'indirizzo virtuale 0xf0000000 e mappando molte altre regioni della memoria virtuale.

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Exercise 3:

While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU monitor commands from the lab tools guide, especially the **xp** command, which lets you inspect physical memory. To access the QEMU monitor, press **Ctrl-a c** in the terminal (the same binding returns to the serial console).
Use the **xp** command in the QEMU monitor and the **x** command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data.
Our patched version of QEMU provides an **info pg** command that may also prove useful: it shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags. Stock QEMU also provides an **info mem** command that shows an overview of which ranges of virtual memory are mapped and with what permissions.

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Nel codice in esecuzione nella CPU, una volta che siamo entrati in modalità protetta (cosa che abbiamo fatto per prima cosa in boot/boot.S), non c'è modo di utilzzare in maniera diretta indirizzi lineari o fisici, in effetti tutti i riferimenti alla memoria sono interpretati come indirizzi virtuali e tradotti dalla MMU e questo significa che tutti i puntatori C sono indirizzi virtuali.

Spesso il kernel JOS ha la necessità di manipolare gli indirizzi in modo opaco, sottoforma di interi, senza dereferenziarli. Questo accade, ad esempio, nel caso del physical memory allocator. Per promuovere la leggibilità nei sorgenti JOS è possibile distinguere diversi casi: **uintptr_t** rappresenta indirizzi virtuali utilizzati in modo opaco, **physaddr_t** rappresenta indirizzi fisici. Entrambi i tipi sono sinonomi di intero a 32 bit (uint32_t) quindi la macchina non ci impedirà di effettuare assegnamenti tra variabili di questi tipi. Ma, dato che sono fondamentalmente degli interi (e non puntatori), il compilatore protesterà se proveremo a dereferenziarli.

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Il kernel JOS potrà dereferenziare un uintptr_t se prima avremmo effettuato il casting ad un tipo valido per un puntatore. Al contrario il kernel non potrà dereferenziare un physaddr_t poiché la MMU traduce tutti i riferimenti alla memoria. Di fatto se effettuiamo il casting di un physaddr_t a puntatore e lo dereferenziamo saremo in grado di leggere e scrivere nella posizione di memoria risultante (lo hardware interpreterà l'indirizzo ottenuto come indirizzo virtuale), ma con ogni probabilità, non agiremo sull'indirizzo di memoria che volevamo utilizzare. Riepilogando:

Tipo C	Tipo di indirizzo
T*	virtuale
uintptr_t	virtuale
physaddr_t	fisico

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Domanda:

1. Assumendo che il seguente codice JOS sia corretto quale è il tipo che dovrebbe avere la variabile x, uintptr_t o physaddr_t?

```
mystery_t x;
   char* value = return_a_pointer();
   *value = 10;
   x = (mystery_t) value;
```

In accordo con l'utilizzo che viene fatto nel codice riportato x, che viene utilizzato per salvare il valore di un puntatore che punta ad un valore (10) x non può essere un physaddr_t, quindi è un **uintptr_t**.

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Il kernel JOS a volte ha la necessità di conoscere e manipolare unicamente indirizzi fisici. Tuttavia in altri casi sia il kernel che altre procedure non possono evitare di passare attravero gli step del meccanismo di traduzione tra i tipi di indirizzi e quindi possono avere a che fare con indirizzi che non sono di tipo fisico. In questi casi JOS inizia dall'indirizzo fisico 0 mappato sull'indirizzo virtuale 0XF0000000 in modo che il kernel possa accedere ad indirizzi fisici di cui già conosce i corrispondenti indirizzi di memoria virtuale. Dato un indirizzo fisico e data la conoscenza dell'indirizzo KERNBASE è possibile tradurre l'indirizzo fisico in un indirizzo virtuale mediante la macro **KADDR** definita in kern/pmap.h .

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

```
00035 /* This macro takes a physical address and returns the corresponding kernel
00036  * virtual address.  It panics if you pass an invalid physical address. */
00037 #define KADDR(pa) _kaddr(__FILE__, __LINE__, pa)
00038
00039 static inline void*
00040 _kaddr(const char *file, int line, physaddr_t pa)
00041 {
00042         if (PGNUM(pa) >= npages)
00043                 _panic(file, line, "KADDR called with invalid pa %08lx", pa);
00044         return (void *)(pa + KERNBASE);
00045 }
```

Notare che è necessaria conoscenza meccanismo di mapping (KERNBASE)

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

In altri casi, dato un indirizzo virtuale, il kernel ha la necessità di conscere il corrispondente indirizzo fisico. Questo può essere realizzato in parte mediante il meccanismo delle page table. Tuttavia esiste un metodo più conveniente di realizzare questa traduzione.
Purchè l'indirizzo virtuale sia al di sopra di KERNBASE possiamo utilizzare la macro **PADDR** definita in pmap.h . Dato il vincolo (vedere sorgente riportato di seguito di passare a PADDR solo indirizzi virtuali al di sopra di KERNBASE) è più corretto dire che essa può tradurre unicamente dei kernel virtual address (piuttosto che parlare di indirizzi virtuali generici).

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

```
00020 /* This macro takes a kernel virtual address -- an address that
points above
00021  * KERNBASE, where the machine's maximum 256MB of physical
memory is mapped --
00022  * and returns the corresponding physical address.  It panics
if you pass it a
00023  * non-kernel virtual address.
00024  */
00025 #define PADDR(kva) _paddr(__FILE__, __LINE__, kva)
00026
00027 static inline physaddr_t
00028 _paddr(const char *file, int line, void *kva)
00029 {
00030         if ((uint32_t)kva < KERNBASE)
00031                 _panic(file, line, "PADDR called with invalid
kva %08lx", kva);
00032         return (physaddr_t)kva - KERNBASE;
00033 }
```

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Page table management:
Ora scriveremo una serie di routine per la gestione delle page table: per inserire e rimuovere mapping da indirizzi lineari ad indirizzi fisici e per creare delle page table in caso di necessità.

Exercise 4. In the file kern/pmap.c, you must implement code for the following functions.

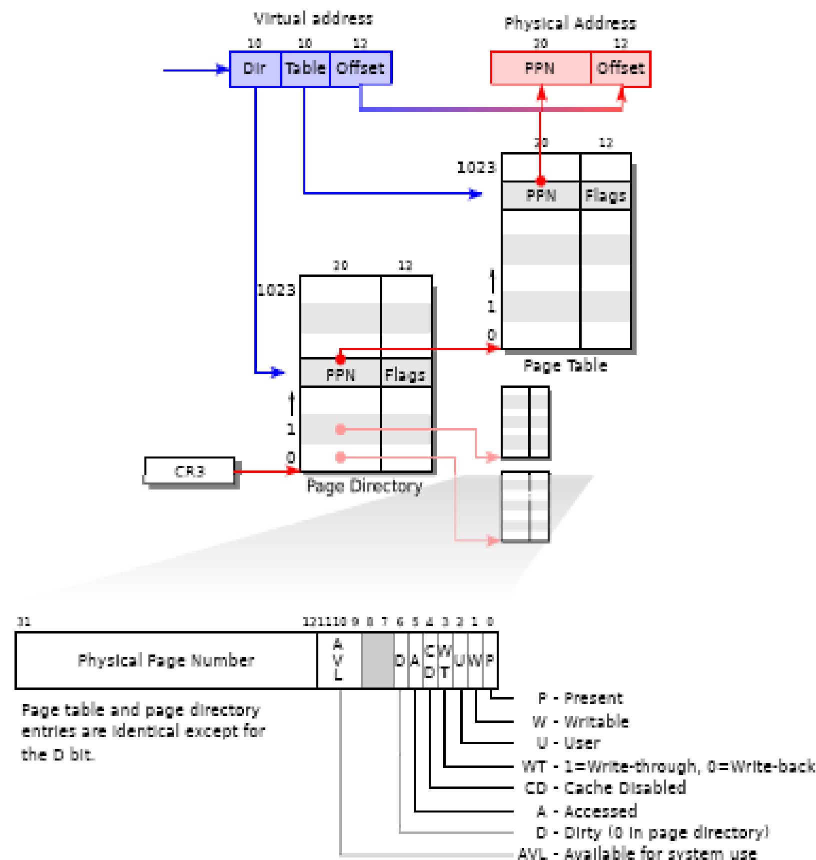
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()

check_page(), called from mem_init(), tests your page table management routines. You should make sure it reports success before proceeding.

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Il processo di traduzione da indirizzo virtuale ad indirizzo fisico (passando per l’indirizzo lineare restituito dal meccanismo di segmentazione) viene realizzato mediante un processo che avviene in due step e di cui di seguito è riportato uno schema:



SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

Scrivere le procedure che realizzano questo processo sarebbe più complesso se i sorgenti JOS non contenessero delle macro che risultano davvero utili. Esse sono definite in **inc/mmu.h** . Facciamo attenzione alla parte in blu (l’indirizzo virtuale). Per operare la traduzione dobbiamo spezzare l’indirizzo virtuale (32 bit) in tre parti: 10 bit alti, 10 bit “intermedi” e 12 bit bassi. In inc/mmu.h sono definite le seguenti macro che permettono di estrarre esattamente queste regioni dall’indirizzo virtuale:

```
00016 // A linear address 'la' has a three-part structure as follows:
00017 //
00018 // +--------10------+-------10-------+---------12----------+
00019 // | Page Directory |   Page Table   | Offset within Page  |
00020 // |      Index     |      Index     |                     |
00021 // +----------------+----------------+---------------------+
00022 //  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
00023 //  \---------- PGNUM(la) ----------/
```

SOLAB2 : MIT JOS lab 2
Memory management

Parte II : Virtual Memory

```
00025 // The PDX, PTX, PGOFF, and PGNUM macros decompose linear addresses as shown.
00026 // To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
00027 // use PGADDR(PDX(la), PTX(la), PGOFF(la)).
00028
00029 // page number field of address
00030 #define PGNUM(la)        (((uintptr_t) (la)) >> PTXSHIFT)
00031
00032 // page directory index
00033 #define PDX(la)          ((((uintptr_t) (la)) >> PDXSHIFT) & 0x3FF)
00034
00035 // page table index
00036 #define PTX(la)          ((((uintptr_t) (la)) >> PTXSHIFT) & 0x3FF)
00037
00038 // offset in page
00039 #define PGOFF(la)        (((uintptr_t) (la)) & 0xFFF)
```

SOLAB2 : MIT JOS lab 2
Memory management

E4.pgdir_walk()
Le specifiche fornite nei sorgenti per questa funzione sono le seguenti:

```
00356 // Given 'pgdir', a pointer to a page directory, pgdir_walk returns
00357 // a pointer to the page table entry (PTE) for linear address 'va'.
00358 // This requires walking the two-level page table structure.
00359 //
00360 // The relevant page table page might not exist yet.
00361 // If this is true, and create == false, then pgdir_walk returns NULL.
00362 // Otherwise, pgdir_walk allocates a new page table page with page_alloc.
00363 //    - If the allocation fails, pgdir_walk returns NULL.
00364 //    - Otherwise, the new page's reference count is incremented,
00365 //      the page is cleared,
00366 //      and pgdir_walk returns a pointer into the new page table page.
00367 //
00368 // Hint 1: you can turn a Page * into the physical address of the
00369 // page it refers to with page2pa() from kern/pmap.h.
00370 //
00371 // Hint 2: the x86 MMU checks permission bits in both the page directory
00372 // and the page table, so it's safe to leave permissions in the page
00373 // more permissive than strictly necessary.
00374 //
00375 // Hint 3: look at inc/mmu.h for useful macros that mainipulate page
00376 // table and page directory entries.
00377 //
```

SOLAB2 : MIT JOS lab 2
Memory management

E4.pgdir_walk()

Qui i suggerimenti su cui basare il lavoro sono quelli di indagare il contenuto di mmu.h (già fatto), di utilizzare **page2pa()** per ottenere l'indirizzo fisico a partire da un puntatore a pagina (abbiamo già parlato di questa funzione) e quello di essere poco stringenti rispetto ai permessi (lo terremo a mente). E' possibile utilizzare anche PTE_ADDR definita in mmu.h riga 76 :

```
00075 // Address in page table or page directory entry
00076 #define PTE_ADDR(pte)    ((physaddr_t) (pte) & ~0xFFF)
```

e KADDR per tradurre un indirizzo virtuale (in spazio kernel) nel corrispondente indirizzo fisico.

SOLAB2 : MIT JOS lab 2
Memory management

E4.pgdir_walk()

In questo esercizio iniziamo a manipolare page directory e page table. Dato un indirizzo lineare vogliamo trovare la corrispondente page directory, l'indirizzo fisico della pagina a cui siamo interessati e l'offset. In definitiva vogliamo avere accesso agli indirizzi in memoria fisica delle pagine. Le entries di page directory e page table sono indirizzi di memoria fisica! Questo significa che, se vogliamo cercare la pagina corrispondente ad un indirizzo in spazio kernel dobbiamo essere in grado di tradurre tra indirizzi fisici e indirizzi virtuali in entrambe le direzioni. Saremo costretti a manipolare entrambi i tipi di indirizzo all’interno della funzione. La conversione da indirizzo virtuale a fisico da operare prima di ritornare il risultato (dopo aver settato i flag in modo da avere permessi più che permissivi) verrà eseguira mediante KADDR.

SOLAB2 : MIT JOS lab 2
Memory management

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
        // Fill this function in
        pte_t * pt;

        if (!(pgdir[PDX(va)] & PTE_P)) { // doesn't exist
                struct PageInfo * p;
                if (!create) return NULL;
                // la creo
                p = page_alloc(ALLOC_ZERO);
                if (!p) return NULL;
                p->pp_ref += 1;
                pgdir[PDX(va)] = page2pa(p)|PTE_P|PTE_W|PTE_U;
        }
        assert(pgdir[PDX(va)] & PTE_P); // either existed or it was created
        pt = (pte_t *)KADDR(PTE_ADDR(pgdir[PDX(va)]));
        return &pt[PTX(va)]; // pt + PTX(VA)*sizeof(pte_t)
}
```

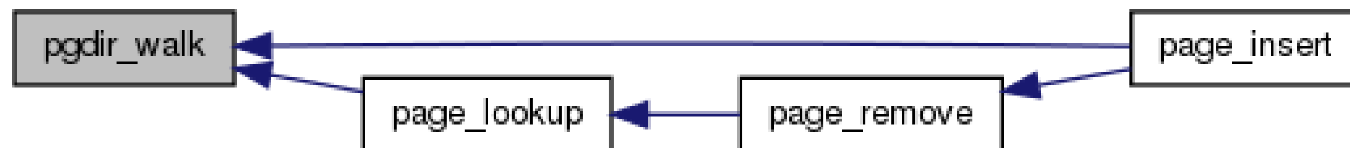
SOLAB2 : MIT JOS lab 2
Memory management

```
pte_t* pgdir_walk ( pde_t *          pgdir,
                    const void *     va,
                    int              create
                  )
```

Definition at line 379 of file pmap.c.

Here is the call graph for this function:

Here is the caller graph for this function:



SOLAB2 : MIT JOS lab 2
Memory management

E4.boot_map_region()

L'obiettivo primario di boot_map_region() è il seguente: data la page directory pgdir, i permessi perm, la dimensione di una pagina PGSIZE, un range definito da due indirizzi virtuali va e va+size la funzione deve mappare il range definito da indirizzi virtuali in un corrispondente range di indirizzi fisici pa, pa+size. La funzione opera al di sopra dell'indirizzo virtuale UTOP (definito in **memlayout.h**) che rappresenta il limite superiore della memoria virtuale che puo essere manipolata in user mode.

```
00118 /*
00119  * Top of user VM. User can manipulate VA from UTOP-1 and down!
00120  */
00121
00122 // Top of user-accessible VM
00123 #define UTOP            UENVS
```

Il tipo di ritorno non è una pagina fisica e quindi non si devono manipolare campi pp_ref (questo è un **range di memoria, non una singola pagina**).

SOLAB2 : MIT JOS lab 2
Memory management

E4.boot_map_region()

Le specifiche nei sorgenti sono queste:

```
00408 // Map [va, va+size) of virtual address space to physical [pa, pa+size)
00409 // in the page table rooted at pgdir.  Size is a multiple of PGSIZE.
00410 // Use permission bits perm|PTE_P for the entries.
00411 //
00412 // This function is only intended to set up the ``static'' mappings
00413 // above UTOP. As such, it should *not* change the pp_ref field on the
00414 // mapped pages.
00415 //
00416 // Hint: the TA solution uses pgdir_walk
```

SOLAB2 : MIT JOS lab 2
Memory management

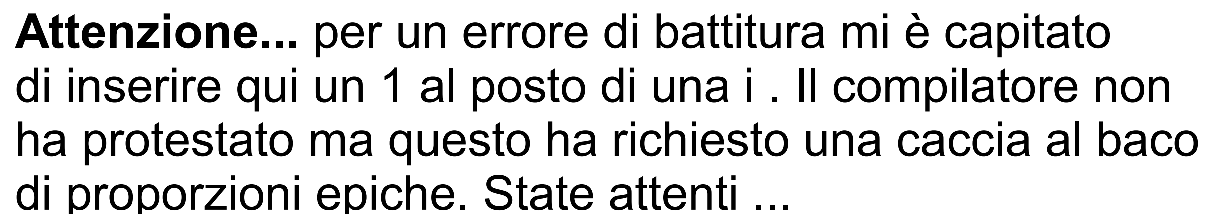
E4.boot_map_region()

Una possibile soluzione è quella di usare un ciclo che attraversa la regione di memoria che va da pa a pa+size utilizzando pgdir_walk() come suggerito dalle specifiche partendo da **pa** e attraversando blocchi di dimensioni multiple di **PGSIZE**. All'interno del ciclo vengono settati i permessi di ogni pagina di memoria fisica. Nel tentativo di non perdere l'allineamento con PGSIZE vengono utilizzate ROUNDUP() e ROUNDDOWN() entrambe definite in inc/types.h .

SOLAB2 : MIT JOS lab 2
Memory management

E4.boot_map_region()

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm
{
	// Fill this function in
	uintptr_t i;
	for (i=0; i<size; i+=PGSIZE) {
		pte_t * ptep = pgdir_walk(pgdir, (void*)(va+i), 1);
		if (ptep)
			*ptep = (pa+i) | perm | PTE_P;
	}
}
```



Attenzione... per un errore di battitura mi è capitato di inserire qui un 1 al posto di una i . Il compilatore non ha protestato ma questo ha richiesto una caccia al baco di proporzioni epiche. State attenti ...

SOLAB2 : MIT JOS lab 2
Memory management

E4.page_lookup()

Data una determinata page directory **pgdir** e un indirizzo virtuale **va** la funzione ritorna un puntatore ad una struct PageInfo associata alla pagina di memoria fisica mappata all'indirizzo va. Se pte_store non è zero allora scriveremo questo valore nella page table entry di questa pagina. Questa funzione è utilizzata da page_remove() e può essere utilizzata (per la verifica dei permessi) dalle chiamate di sistema. La funzione ritorna NULL se non esiste una pagina che mappa in va.

Le specifiche fornite nei commenti presenti nei sorgenti sono le seguenti:

```
00490 // Return the page mapped at virtual address 'va'.
00491 // If pte_store is not zero, then we store in it the address
00492 // of the pte for this page.  This is used by page_remove and
00493 // can be used to verify page permissions for syscall arguments,
00494 // but should not be used by most callers.
00495 //
00496 // Return NULL if there is no page mapped at va.
00497 //
00498 // Hint: the TA solution uses pgdir_walk and pa2page.
```

SOLAB2 : MIT JOS lab 2
Memory management

E4.page_lookup()

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
        // Fill this function in
        pte_t * ptep = pgdir_walk(pgdir, va, 0);
        if (ptep) {
                physaddr_t pa;
                assert(*ptep & PTE_P); // mapped
                pa = PTE_ADDR(*ptep);
                if (pte_store != 0)
                        *pte_store = ptep;
                return pa2page(pa);
        }
        // not mapped
       return NULL;
}
```

SOLAB2 : MIT JOS lab 2
Memory management

E4.page_remove()

Lo scopo di questa funzione è rimuovere l'associazione tra l'indirizzo virtuale **va** ed una **pagina di memoria fisica** (nel caso in cui tale associazione esista).

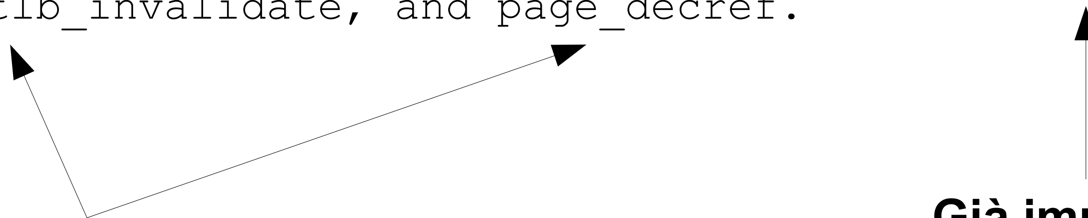
Nel caso in cui nessuna pagina di memoria fisica sia mappata all'indirizzo virtuale va ritorna senza fare nulla. Nel caso in cui l'associazione esista la funzione deve decrementare il numero di riferimenti alla pagina di memoria fisica e, nel caso in cui questo valore sia zero dopo il decremento, deve reinserire la pagina nella lista di pagine di memoria libere. Nel caso in cui l'associazione tra va e pagina sia stata rimossa dalla funzione quest'ultima deve occuparsi di impostare a **0** il valore della page table entry corrispondente a va. Se un elemento viene rimosso dalla page table TBL deve essere invalidata. Il tipo di ritorno è void.

SOLAB2 : MIT JOS lab 2
Memory management

E4.page_remove()

Specifiche:

```
00519 //
00520 // Unmaps the physical page at virtual address 'va'.
00521 // If there is no physical page at that address, silently does nothing.
00522 //
00523 // Details:
00524 //   - The ref count on the physical page should decrement.
00525 //   - The physical page should be freed if the refcount reaches 0.
00526 //   - The pg table entry corresponding to 'va' should be set to 0.
00527 //     (if such a PTE exists)
00528 //   - The TLB must be invalidated if you remove an entry from
00529 //     the page table.
00530 //
00531 // Hint: The TA solution is implemented using page_lookup,
00532 //       tlb_invalidate, and page_decref.
00533 //
```



SOLAB2 : MIT JOS lab 2
Memory management

E4.page_remove()

tbl_invalidate (in kern/pmap.c):

```
00554 // Invalidate a TLB entry, but only if the page tables being
00555 // edited are the ones currently in use by the processor.
00556 //
00557 void
00558 tlb_invalidate(pde_t *pgdir, void *va)
00559 {
00560         // Flush the entry only if we're modifying the current address
space.
00561         // For now, there is only one address space, so always
invalidate.
00562         invlpg(va);
00563 }
```

SOLAB2 : MIT JOS lab 2
Memory management

E4.page_remove()

page_decref (in kern/pmap.c):

```
00346 // Decrement the reference count on a page,
00347 // freeing it if there are no more refs.
00348 //
00349 void
00350 page_decref(struct PageInfo* pp)
00351 {
00352         if (--pp->pp_ref == 0)
00353                 page_free(pp);
00354 }
```

Come possiamo vedere page_decref() si occupa sia di decrementare il numero di riferimenti ad una pagina che di renderla nuovamente disponibile (chiamando page_free()) nel caso in cui il numero di riferimenti alla pagina sia 0.

SOLAB2 : MIT JOS lab 2
Memory management

E4.page_remove()

```
void
page_remove(pde_t *pgdir, void *va)
{
        // Fill this function in
        pte_t * ptep;
        struct PageInfo * page = page_lookup(pgdir, va, &ptep);
        if (page) {
                assert(page->pp_ref > 0);
                page_decref(page);
                *ptep = 0;
                tlb_invalidate(pgdir, va);
        }
}
```

SOLAB2 : MIT JOS lab 2
Memory management

E4.page_insert()

Questa funzione serve ad inserire in una page directory una nuova associazione tra una pagina di memoria fisica e l'indirizzo di memoria virtuale va passato come argomento unitamente ai permessi perm e ad un puntatore a struct PageInfo. Le specifiche riportate nei sorgenti sono le seguenti:

```
00441 // Map the physical page 'pp' at virtual address 'va'.
00442 // The permissions (the low 12 bits) of the page table entry
00443 // should be set to 'perm|PTE_P'.
00444 //
00445 // Requirements
00446 //   - If there is already a page mapped at 'va', it should be page_remove()d.
00447 //   - If necessary, on demand, a page table should be allocated and inserted
00448 //     into 'pgdir'.
00449 //   - pp->pp_ref should be incremented if the insertion succeeds.
00450 //   - The TLB must be invalidated if a page was formerly present at 'va'.
00451 //
00452 // Corner-case hint: Make sure to consider what happens when the same
00453 // pp is re-inserted at the same virtual address in the same pgdir.
00454 // However, try not to distinguish this case in your code, as this
00455 // frequently leads to subtle bugs; there's an elegant way to handle
00456 // everything in one code path.
00457 //
00458 // RETURNS:
00459 //   0 on success
00460 //   -E_NO_MEM, if page table couldn't be allocated
00461 //
00462 // Hint: The TA solution is implemented using pgdir_walk, page_remove,
00463 // and page2pa.
```

SOLAB2 : MIT JOS lab 2
Memory management

E4.page_insert()

Il tipo di ritorno è un intero. Sarà 0 in caso di successo e –E_NO_MEM se la pagina non può essere allocata. Le specifiche sono molto ben definite. Viene suggerito di utilizzare pgdir_walk(), page_remove() e page2pa.

```
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
	// Fill this function in
	pte_t * ptep = pgdir_walk(pgdir, va, 1);
	if (!ptep) return -E_NO_MEM;
	if (*ptep & PTE_P) { // already mapped (PTE_P in PT entry)
		if (PTE_ADDR(*ptep) == page2pa(pp)) { // but not the same
			*ptep = page2pa(pp) | perm | PTE_P;
			return 0;
		}
		page_remove(pgdir, va);
	}
	// only here and boot_map_region
	// PTE_P (in PT entry) is set
	*ptep = page2pa(pp) | perm | PTE_P;
	pp->pp_ref += 1;
	return 0;
}
```

Questo conclude Esercizio 4

SOLAB2 : MIT JOS lab 2
Memory management

Parte III : Kernel Address Space

JOS divide lo spazio degli indirizzi a 32 bit lineari del processore in due parti. Gli *user environments* (i processi) di cui ci occuperemo in Lab3 hanno il controllo sul layout ed il contenuto della parte bassa mentre il kernel maniene sempre il controllo completo della parte alta. La linea di divisione tra queste due aree è impostata, arbitrariamente, in corrispondenza del simbolo **ULIM** definito in inc/memlayout.h :

```
#define MMIOLIM          (KSTACKTOP - PTSIZE)
#define MMIOBASE         (MMIOLIM - PTSIZE)

#define ULIM             (MMIOBASE)
```

Questo riserva approssimativamente uno spazio di indirizzamento di **256 MB** (indirizzi virtuali) per il kernel. Questo spiega come main, in Lab1, abbiamo dovuto dare al kernel un link address così alto. In caso contrario non ci sarebbe stato abbastanza posto nello spazio degli indirizzi virtuali del kernel per mappare in un user environment posto al di sotto di esso.

SOLAB2 : MIT JOS lab 2
Memory management

Per questa parte del laboratorio può essere utile fare riferimento allo schema del layout della memoria di JOS presente in **inc/memlayout.h**

```
00022  * Virtual memory map:                                Permissions
00023  *                                                    kernel/user
00025  *    4 Gig -------->  +------------------------------+
00026  *                     |                              | RW/--
00027  *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
00028  *                     :              .               :
00030  *                     :              .               :
00031  *                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
00032  *                     |                              | RW/--
00033  *                     |   Remapped Physical Memory   | RW/--
00034  *                     |                              | RW/--
00035  *    KERNBASE, ---->  +------------------------------+ 0xf0000000      --+
00036  *    KSTACKTOP        |     CPU0's Kernel Stack      | RW/--  KSTKSIZE   |
00037  *                     | - - - - - - - - - - - - - - -|                   |
00038  *                     |      Invalid Memory (*)      | --/--  KSTKGAP    |
00039  *                     +------------------------------+                   |
00040  *                     |     CPU1's Kernel Stack      | RW/--  KSTKSIZE   |
00041  *                     | - - - - - - - - - - - - - - -|                 PTSIZE
00042  *                     |      Invalid Memory (*)      | --/--  KSTKGAP    |
00043  *                     +------------------------------+                   |
00044  *                     :              .               :                   |
```

```
00046  *    MMIOLIM ------>  +------------------------------+ 0xefc00000      --+
00047  *                     |       Memory-mapped I/O      | RW/--  PTSIZE
00048  * ULIM, MMIOBASE -->  +------------------------------+ 0xef800000
00049  *                     |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
00050  *    UVPT      ---->  +------------------------------+ 0xef400000
00051  *                     |          RO PAGES            | R-/R-  PTSIZE
00052  *    UPAGES    ---->  +------------------------------+ 0xef000000
00053  *                     |           RO ENVS            | R-/R-  PTSIZE
00054  * UTOP,UENVS ------>  +------------------------------+ 0xeec00000
00055  * UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
00056  *                     +------------------------------+ 0xeebff000
00057  *                     |       Empty Memory (*)       | --/--  PGSIZE
00058  *    USTACKTOP  --->  +------------------------------+ 0xeebfe000
00059  *                     |      Normal User Stack       | RW/RW  PGSIZE
00060  *                     +------------------------------+ 0xeebfd000
00061  *                     |                              |
00062  *                     |                              |
00063  *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
00064  *                     .                              .
00066  *                     .                              .
00067  *                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
00068  *                     |     Program Data & Heap      |
00069  *    UTEXT -------->  +------------------------------+ 0x00800000
00070  *    PFTEMP ------->  |       Empty Memory (*)       |        PTSIZE
00071  *                     |                              |
00072  *    UTEMP -------->  +------------------------------+ 0x00400000      --+
00073  *                     |       Empty Memory (*)       |                   |
00074  *                     | - - - - - - - - - - - - - - -|                   |
00075  *                     |  User STAB Data (optional)   |                 PTSIZE
00076  *    USTABDATA ---->  +------------------------------+ 0x00200000        |
00077  *                     |       Empty Memory (*)       |                   |
00078  *    0 ------------>  +------------------------------+                 --+
00080  * (*) Note: The kernel ensures that "Invalid Memory" is *never* mapped.
00081  *     "Empty Memory" is normally unmapped, but user programs may map pages
00082  *     there if desired.  JOS user programs map pages temporarily at UTEMP. */
```

SOLAB2 : MIT JOS lab 2
Memory management

Permissions and Fault Isolation

Dato che gli spazi di indirizzamento di ogni user environment contengono sempre sia memoria kernel che memoria utente dobbiamo utilizzare in modo corretto i bit delle nostre x86 page tables per impostare i permessi in modo da permettere al codice utente di accedere solamente alla parte dello spazio di indirizzamento ad esso destinata. In caso contratio bachi nel codice utente potrebbero sovrascrivere i dati del kernel causando un crash di sistema o problemi meno evidenti (ma non per questo meno importanti) per la coerenza dello stato del sistema. L’assenza di una impostazione corretta dei permessi potrebbe anche dare la possibilità al codice utente di accedere ai dati di altri user environments compromettendo la sicurezza di informazioni sensibili.

User environment non avrà alcun permesso a nessun indirizzo di memoria superiore a ULIM mentre il kernel sarà in grado di leggere e scrivere questa area di memoria. Nel range di indirizzi [UTOP, ULIM) sia il kernel che lo user environment avranno gli stesis permessi: entrambi potranno **leggere ma non scrivere in questo range di indirizzi**. Questa area di memoria è utilizzata per esporre (in modo read-only) alcune strutture dati del kernel allo user environment. Infine lo spazio al di sopra di UTOP è a disposizione dello user environment.

SOLAB2 : MIT JOS lab 2
Memory management

Initializing the Kernel Address Space

In questa parte del laboratorio dovremo impostare lo spazio degli indirizzi **al di sopra** di **UTOP** : la parte dello spazio di indirizzamento dedicata al kernel. Il layout da utilizzare è mostrato in **inc/memlayout.h** . Utilizzeremo le funzioni che abbiamo implementato in precedenza per stabilire un mapping appropriato tra indirizzi **lineari e fisici**.

Exercise 5 (modificare sorgente pmap.c) :
Fill in the missing code in **mem_init()** after the call to **check_page()**. Your code should now pass the check_kern_pgdir() and check_page_installed_pgdir() checks.

La combinazione delle informazioni disponibili dello schema (incluso in questo documento) del layout della memoria presente in **inc/memlayout.h** e delle informazioni fornite nei commenti presenti nei sorgenti rende la soluzione di questo esercizio decisamente abbordabile. Riporto la parte di codice interessata con le relative soluzioni proposte.

SOLAB2 : MIT JOS lab 2
Memory management

Initializing the Kernel Address Space

Innanzitutto notiamo che dopo la chiamata a check_page() ci sono 3 spazi contrassegnati con “your code goes here”. Partiamo dal primo:

```
//////////////////////////////////////////////////////////////////////
// Now we set up virtual memory

//////////////////////////////////////////////////////////////////////
// Map 'pages' read-only by the user at linear address UPAGES
// Permissions:
//    - the new image at UPAGES -- kernel R, user R
//      (ie. perm = PTE_U | PTE_P)
//    - pages itself -- kernel RW, user NONE
// Your code goes here:

// PTE_P from boot_map_region
boot_map_region(kern_pgdir, UPAGES, sizeof(struct PageInfo)*npages,
                PADDR(pages), PTE_U);
```

Il 2° ...

```
//////////////////////////////////////////////////////////////////////
// Use the physical memory that 'bootstack' refers to as the kernel
// stack.  The kernel stack grows down from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
// to be the kernel stack, but break this into two pieces:
//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
//     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
//       the kernel overflows its stack, it will fault rather than
//       overwrite memory.  Known as a "guard page".
//     Permissions: kernel RW, user NONE
// Your code goes here:

// PTE_P from boot_map_region
boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE,
                PADDR(bootstack), PTE_W);
// PTE_P must not be set (not backed)
// thus boot_map_region is not useful
for (n=0; n<PTSIZE-KSTKSIZE; n+=PGSIZE){
        pte_t *ptep = pgdir_walk(kern_pgdir,
                                 (void*)(KSTACKTOP-PTSIZE+n), 1);
        if (ptep)
                *ptep = 0;
}
```

NB

SOLAB2 : MIT JOS lab 2
Memory management

Initializing the Kernel Address Space

Infine il terzo:

```
//////////////////////////////////////////////////////////////////////
// Map all of physical memory at KERNBASE.
// Ie.  the VA range [KERNBASE, 2^32) should map to
//      the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:

boot_map_region(kern_pgdir, KERNBASE, 0xffffffff-KERNBASE+1, 0, PTE_W);
```

Questo completa esercizio 5. Provate a fare make grade ...

SOLAB2 : MIT JOS lab 2
Memory management

Vediamo se la soluzione proposta è accettata da **make grade:**

```
+ 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: (12.6s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
user@:/home/jos/solab-jos$ _
```