APPUNTI PER IL CORSO DI SISTEMI OPERATIVI

A.A. 2013 2014

PARTE 1: BOOTSTRAP E BOOTLOADER

VERSIONE BETA

A cura di:

Danilo Bruschi
Roberto Stucchi

Copyright: in questi appunti si descrivono e commentano porzioni di codice del sistema operativo JOS di cui riportiamo la relativa nota di copyright:

Il bootstrap

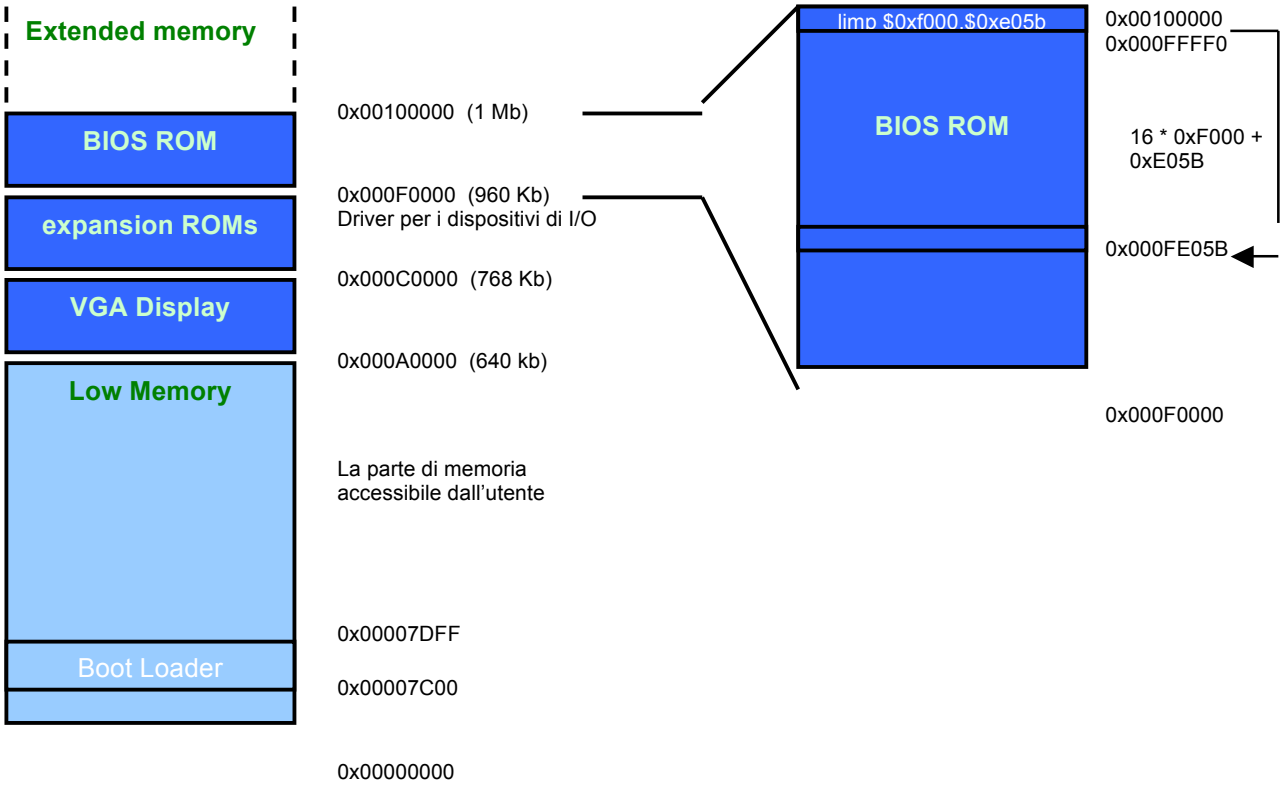
I dettagli definiti in questo paragrafo fanno riferimento all'architettura Intel 8088 e pur rimandendo concettualmente identici, i loro dettagl iimplementatitivi sono stati nel tempo modificati per adattarsi all'evoluzione delle diverse architettetture.

Vediamo prima brevemente come funziona il bootstrap di un sistema, facendo riferimento all'architettura Intel 8088. Quando si accende un PC, il microprocessore resta in attesa di un segnale di power good che gli sarà fornito quando tutti i circuiti non sono stati alimentati correttamente. Più precisamente, il segnale di "power good" è ricevuto dal clock (timer chip) che nel frattempo continua a resettare il processore non consentogli l'avvio. Alla ricezione del segnale di Power Good il clock asserisce il pin RESET# sul processore che avvia la fase di inizializzazione.
In questa fase il processore inizializza i registri a dei valori predeterminati, ci interessano per ora solo i valori dei registri CS : 0xf000 ed IP: 0xfff0. Questi due valori infatti determinano l'indirizzo della prima istruzione che sarà eseguita dal processore, valore che si ottiene applicando la seguente formula:

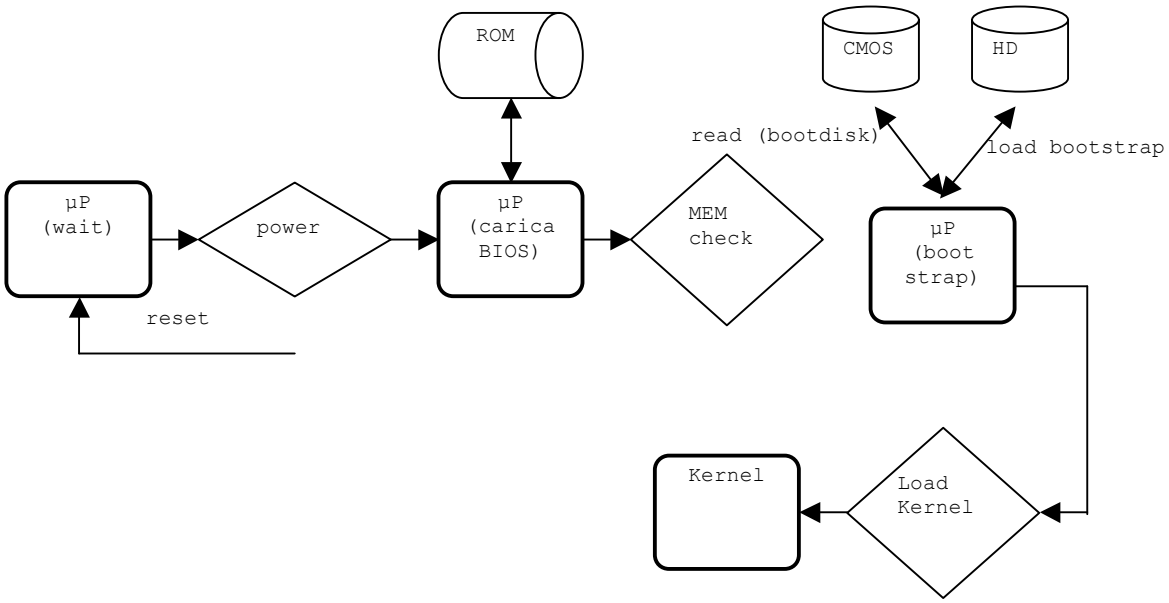
16 * CS + IP

Quindi nel nostro caso l'indirizzo ottenuto è 0xffff0, che è un indirizzo del BIOS (gli indirizzi del BIOS sono compresi tra 0xf0000 e 0xfffff) in cui per convenzione viene caricata la prima istruzione che sarà eseguita dal processore che è l'istruzione:

```
ljmp $0xf000,$0xe05b
```



L'istruzione punta quindi ad un indirizzo di memoria (0xfe05b) sempre relativo al BIOS che è di fatto un programma di sistema il cui compito è verificare il corretto funzionamento dei dispositivi che compongono il sistema e caricare in memoria il boot loader. Più precisamente il BIOS inizializza tutti i dispositivi di input/output (tra cui anche il controller IDE del disco) ed esegue un test della memoria RAM detto POST, che viene eseguito ogni volta che viene fatto un riavvio hardware (cold boot) del PC. Una volta terminata questa fase, il BIOS si preoccupa dall'area CMOS i rifierimenti del dispositivo di boot cioè del dispositivo da cui caricare il boot loader che a sua volta si occuperà del caricamento del Kernel. Il boot loader la cui dimensione è limitata è nel nostro caso caricato nel primo settore del disco (detto boot sector 512 byte) ed è caricato dal BIOS in RAM nell'area compresa tra gli indirizzi 0x7c00 e 0x7dff, terminato il caricamento il BIOS setta setta CS:IP a 0x0000:0x7c00, (cioè l'indirizzo della prima istruzione del bootloader) a cui cede il controllo, da questo momento il BIOS non sarà più utilizzato, sino al prossimo riavvio del sistema.



IL BOOTLOADER

Compito fondamentale del bootloader è caricare in memoria centrale il Kernel del sistema operativo, per fare quest'operazione deve però predisporre l'hardware a poter ospitare questo programma, in particolare deve predisporre il sistema ad operare in protected mode per poter avere visibilità dell'intera memoria. La parte di memoria che va dal 1Mb ai 4Gb viene detta extended memory, ed in JOS il kernel sarà caricato a partire dall'indirizzo fisico 0x00100000. Quest'operararazione deve però essere preceduta da alcune attività preliminari, prima fra tutte l'abilitazione della linea 21 del bus indirizzi (A 20). Questo il macro schema del boot loader:

```
{
enable A20 line;
enable 32-bit protected mode;
read kernel from disk;
jmp to first kernel instruction;
}
```

Abilitazione linea A20

Per garantire la back compatibilità con architetture precedenti la linea A20 del bus è disabilitatas in fase di boot. Questo consente in real mode di avere solo indirizzi inferiori a 0xFFFFF e quindi esprimibili con 20 bit. Nel momento in cui ci si one il

problema di estendere lo spazio degli indirizzi da 20 a 32 bit e necessario come prima cosa abilitare la suddetta linea.
Il metodo tradizionale per l'abilitazione delle linea A20 prevede di operare direttamente sul controller della tastiera, questo perché in fase progettuale il controller della tastiera 8042 di Intel, aveva un pin inutilizzato che si è pensato di usare per controllare lo stato della liena A20. Il valore di questo pin viene settato predispondendo il secondo bit della porta di uscita del controller della tastiera, in particolare se questo bit vale zero il 21 bit del bus indirizzi è sempre zero, se vale 1 il 21 bit manteràà il valore che gli è stato originariamente assegnato.

Per scrivere sulla porta di uscita (indirizzo 0x60) del controller 8042 devono essere svolte queste operazioni :

1. Deve essere inviato un comando "Write output port" cioè 0xD1 al registro comandi del controller, questo significa che 0xD1 deve essere "out" alla porta 0x64,
2. I dati da inserire nella porta di uscita devono essere scritti sulla porta 0x60.

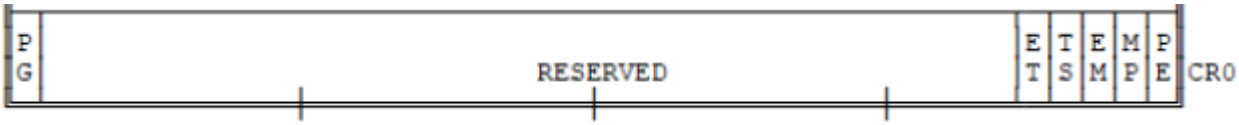
Prima di eseguire i comandi di cui sopra, un ciclo per verificare se il controller della kyboard è pronto a ricevere i comandi. Qui di seguito il codice assembler per eseguire la suddetta funzione:

```
# Enable A20:
   #   For backwards compatibility with the earliest PCs, physical
   #   address line 20 is tied low, so that addresses higher than
   #   1MB wrap around to zero by default.  This code undoes this.

 seta20.1:
   inb     $0x64,%al               # Wait for not busy
   testb   $0x2,%al
   jnz     seta20.1

   movb    $0xd1,%al               # 0xd1 -> port 0x64
   outb    %al,$0x64

 seta20.2:
   inb     $0x64,%al               # Wait for not busy
   testb   $0x2,%al
   jnz     seta20.2

   movb    $0xdf,%al               # 0xdf -> port 0x60
   outb    %al,$0x60
```

Abilitazione a 32-bit protected mode

Inizialmente il boot loader abilita il protected mode con la sola segmentazione, la paginazione verrà abilitata più avanti dal kernel con la memoria virtuale. Per attivare questa modalità è necessario:

- predisporre una tabella dei segmenti (GDT) che contenga almeno i descrittori dei segmenti codice e dati relativi al kernel, specificando nella struttura segdesc il campo sd_db a 1 che indica che i segmenti in questione sono relativi ad una modalità a 32-bit,
- caricare l'indirizzo di questa tabella in GDTR,
- settare a uno il bit 0 del registro Cr0
- sovrascrivere il registro CS con il descrittore del segmento codice caricato in GDT.



Caricamento GDT

Queste le itruzioni presenti nel boot loader che provvedono a dichiarare e inizalizzare la GDT:

```
# Bootstrap GDT
.p2align 2                # force 4 byte alignment
# indirizzo gdt contiene 3 descrittori di segmento, ogni
# descrittore è lungo 8 byte
gdt:
  SEG_NULL                          # segmento nullo
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # segmento codice
  SEG(STA_W, 0x0, 0xffffffff)       # segmento dati

# Il campo da caricare in GDTR è formato da 6 byte. I primi 2
# indicano la dimensione della GDT. Il secondo l'indirizzo

gdtdesc:
  .word   0x17                              # sizeof(gdt) - 1
  .long   gdt                               # address gdt
```

Dove per le macro e le costanti valgono le seguenti definizioni:

```
/* Macros to build GDT entries in assembly */

 #define SEG_NULL                                \
     .word 0, 0;                                 \
     .byte 0, 0, 0, 0

 #define SEG(type,base,lim)                                \
     .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);    \
     .byte (((base) >> 16) & 0xff), (0x90 | (type)),       \
          (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)

 // Application segment type bits

 #define STA_X      0x8      // Executable segment
 #define STA_W      0x2      // Writeable (non-executable segments)
 #define STA_R      0x2      // Readable (executable segments)
 #define STA_A      0x1      // Accessed
```

Le istruzioni che provvedono invece a caricare il registro GDTR e il registro CR0 con i dati necessari ad eseguire la modifica di stato sono:

```
  lgdt    gdtdesc             # carica GDTR
  movl    %cr0, %eax          # carico il contenuto di cr0 in eax
  orl     $CR0_PE_ON, %eax    # setta a 1 bit 0 di %eax
  movl    %eax, %cr0          # muovo il valore di %eax in cr0

  # A questo punto non siamo ancora in protected mode, bisogna
  # caricare nel registro CS il descrittore di un segmento
  # a 32 bit, per modificare CS usiamo l'istruzione ljmp

  ljmp    $PROT_MODE_CSEG, $protcseg
```

Leggere il kernel da disco

Questa attività viene svolta dal boot loader richiamando una routine scritta in C, presente nel file /lab/boot/main.c che sarà spiagata nella prossima sezione.

Questo il listato completo del boot loader che ripetiam viene letto dal BIOS dal boot sector e caricato in memoria a partire dall'indirizzo 0x007c

File: boot.s

```
#include <inc/mmu.h>

.set PROT_MODE_CSEG, 0x8             # kernel code segment selector
```

```
.set PROT_MODE_DSEG, 0x10            # kernel data segment selector
.set CR0_PE_ON,      0x1             # protected mode enable flag

.globl start
start:                 # start è entry pointer del codice
  .code16              # segmento codice a 16 bit

  cli                  # Disabilita interrupt
  cld                   /* istruzione che serve per indicare l'incremento,
                         il decremento sulle operazioni di stringhe (es:
                         movs che usa come puntatore di inizio della
                         stringa il registro EDI e come contatore il
                         registro ESI) il contatore può andare in
                         incremento o in decremento in funzione del
                         valore di cld quando arriva a 0 si ferma. */

/* Azzera i segment register. I segment register possono essere
modificati con i valori di altri registri */

  xorw    %ax,%ax              # Azzera segment register
  movw    %ax,%ds              # -> Data Segment
  movw    %ax,%es              # -> Extra Segment
  movw    %ax,%ss              # -> Stack Segment


  # ABILITA A20

seta20.1:
  inb     $0x64,%al   # porta 64 è pronta a ricevere ?
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al   # 0xd1 (comando di scrittura) -> %al
  outb    %al,$0x64   # commando di scrittura porta 0x64

seta20.2:
  inb     $0x64,%al   # Verifico con lo stesso ciclo se il
  testb   $0x2,%al    # dispositivo è pronto
  jnz     seta20.2    # Salta se non è zero

  movb    $0xdf,%al   # 0xdf è il valore da mandare
  outb    %al,$0x60   # sulla porta 0x60


# Fase per passare da real mode a protected mode.
# gdt e gdtdesc sono aree dati del bootloader

  lgdt    gdtdesc              # carica la GDT
  movl    %cr0, %eax       # carico il contenuto di cr0 in eax
  orl     $CR0_PE_ON, %eax # eseguo OR con la costante
  movl    %eax, %cr0       # muovo il valore di eax in cr0
                           # il motivo per cui vengono fatti questi 3
                           # comandi invece di fare una
                           # movl $1, %eax
                           # movl %eax, %cr0
                           # è per preservare il contenuto di cr0
```

```
# A questo punto non siamo ancora in protected mode, per farlo occorre
# eseguire una ljmp a un segment descriptor che sia relativo a un
# segmento a 32 bit.
# Il comando ljmp carica un dato in CS e poi ci aggiunge lo spiazzamento.

  ljmp    $PROT_MODE_CSEG, $protcseg  # Prendi il segmento spiazzato di 8
                                      # rispetto all'indirizzo che trovi
                                      # in gdtr e somma protcseg che è
                                      # l'indirizzo successivo.
                                      #
# Questa è l'ultima istruzione eseguita in real mode.
```

GDT

(indirizzo) 0	SEG_NULL	segmento nullo
(indirizzo) 8	SEG(STA_X|STA_R, 0x0, 0xffffffff)	segmento codice
(indirizzo) 16	SEG(STA_W, 0x0, 0xffffffff)	segmento dati

```
# A questo punto il processore opera in protected mode a 32 bit

  .code32                     # Assemble for 32-bit mode

# inizializza segment register con l'inidirizzo del Data segment

protcseg:
  movw    $PROT_MODE_DSEG, %ax    # segmento dati
  movw    %ax, %ds
  movw    %ax, %es
  movw    %ax, %fs
  movw    %ax, %gs
  movw    %ax, %ss

  # Adesso quello che ci resta da fare è caricare il kernel.
  # I sistemisti di JOS hanno pensato di farlo con una routine in C.
  # Si crea uno stack che verrà usato successivamente dalla routine C
  # 'bootmain'.

  movl    $start, %esp   # start = 0x00007c00 è l'indirizzo
                         #  della prima istruzione eseguibile del
                         #  bootloader

  # bootmain è una procedura che legge il kernel
  # da disco e lo copia in memoria e poi fa partire il kernel,
  # quindi il bootloader non rientra mai

  call bootmain

  # Il bootmain non dovrebbe mai rientrare altrimenti va in loop

spin:
  jmp spin


# Bootstrap GDT
.p2align 2            # force 4 byte alignment
                      # gdt contiene 3 descrittori di segmento,
                      # ogni descrittore è lungo 8 byte
gdt:
```

```
  SEG_NULL                              # segmento nullo
  SEG(STA_X|STA_R, 0x0, 0xffffffff)     # segmento codice
                                        # dimensione 4Gb
  SEG(STA_W, 0x0, 0xffffffff)           # segmento dati

gdtdesc:   # Campo formato da 6 byte da caricare in GDTR

  .word   0x17                              # sizeof(gdt) - 1
  .long   gdt                               # address gdt
```

APPUNTI PER IL CORSO DI SISTEMI OPERATIVI

A.A. 2013/2014

PARTE 2: Il CARICAMENTO DEL KERNEL

A cura di:

Danilo Bruschi
Roberto Stucchi

BOOTMAIN

Bootmain è l'netry point della procedura C che ha come scopo quello di leggere da disco il kernel che è stato memorizzato in formato ELF. Il formato ELF è stato adottato da diversi sistemi per la memorizzazione di programmi in formato oggetto o eseguibile, e sarà illustrato successivamente.
Ci preoccupiamo invece ora di analizzare le modalità con cui la procedura bootmain legge e trasferisce il kernel dal disco alla memoria.
La procedura bootmain assume che il disco di riferimento abbia un interfaccia IDE e il controller adotti come modalità di indirizzamento dei blocchi del disco la modalità LBA mode che prevede che il disco sia visto come un array consecutivo di blocchi da 512 byte, ciascuno indirizzabile da un numero intero di 28bit. Questa scelta viene fatta dal BIOS durante l'inizializzazione. Più precisamente, il metodo LBA, introdotto con lo standard ATA-2, i settori del disco sono numerati da 0 a $2^{28} - 1$, assegnando il valore 0 al primo settore della prima traccia del primo cilindro, procedendo poi lungo tutti i settori della stessa traccia, poi lungo tutte le tracce (corrispondenti a tutte le superfici) dello stesso cilindro per poi spostarsi al cilindro adiacente, continuando così fino all'ultimo settore dell'ultima traccia dell'ultimo cilindro. La comunicazione con il controlle IDE avviene utilizzando le seguenti porte ed i relativi indirizzi:

0x1F0	Data Port (risultato lettura dati)
0x1F1	Error (segnalazione errori)
0x1F2	Sector Count (numero dei settori da leggere)
0x1F3	LBA Low byte
0x1F4	LBA mid byte
Ox1F5	LBA hi byte
0x1F6	1B1D TOP4LBA: B=LBA, D=driv (Drive e testina)
0x1F7	Command/Status (Comando / Stato della periferica)

indirizzo LBA mode (28 bit)

Status Register (0x1F7)							
7	6	5	4	3	2	1	0
BUSY	READY	FAULT	SEEK	DRO	CORR	IDDEX	ERROR

Error Register (0x1F1)							
7	6	5	4	3	2	1	0
BBK	UNC	MC	IDNF	MCR	ABRT	TONF	AMNF

BBK = Bad Block
UNC = Uncorrectable data error

MC = Media changed
IDNF = ID mark not found
MCR = Media Change Requested
ABRT = Command Aborted
TONF= Track 0 Not Found
AMNF = Address Mark Not Found

Come si usa un dispositivo IDE

Descriviamo brevement quali sono i passi da intraprendere per poter utilizzare un dispositivo IDE.

1. Prima eseguire un qualunque operazione bisogna aspettare che il controller sia in uno stato di ready (bit RDY nel registro di stato)
2. Si caricano i parametri del comando che si vuole eseguire nei registri appositi. Per i comandi di lettura / scrittura significa scrivere nei registri l'inidirizzo del settore interessato
3. Si inoltra un comando di lettura o scrittura.
4. Si attende fino ai segnali del dispositivo che è pronto per il trasferimento dati (DRQ nel registro di stato).
5. Alimentare i dati del dispositivo (per la scrittura) o ottenere i dati dal dispositivo (per la lettura).
6. In caso di una scrittura si può attendere il completamento dell'operazione e leggere dal registro di stato l'esito dell'operazione.

I principali comandi sono:

- 20H: Leggi il settore con retry. NB: 21H = lettura senza retry. Per questo comando è necessario caricare preventivamente l'indirizzo del settore che si vuole leggere. Una volta completato il comando (DRQ viene attivato) si possono leggere 256 word dal registro dati del disco.
- 30H: Scrivere un settore con retry, 31H = senza tretry. Anche in quesot caso è necessario caricare preventivamente l'indirizzo del settore su cui si vuole scrivere. Quindi attendere che DRQ diventi attivo e alimentare il disco attraverso il dta register con 256 word di dati. Successivamente il disco inizia a scrivere. Quando BSY va a zero si può leggere lo stato dal registro di stato.

Più in dettagli questi sono i dati da inviare al controller IDE per leggere un settore da disco:

- Send a NULL byte to port 0x1F1: outb(0x1F1, 0x00);
- Send a sector count to port 0x1F2: outb(0x1F2, 0x01);

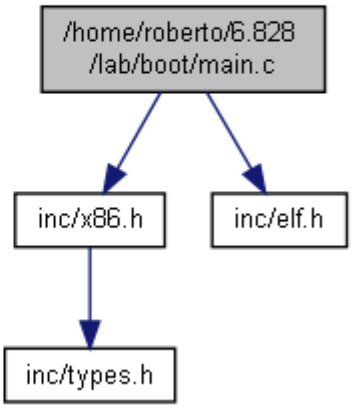
- Send the low 8 bits of the block address to port 0x1F3: outb(0x1F3, (unsigned char)addr);
- Send the next 8 bits of the block address to port 0x1F4: outb(0x1F4, (unsigned char)(addr >> 8);
- Send the next 8 bits of the block address to port 0x1F5: outb(0x1F5, (unsigned char)(addr >> 16);
- Send the drive indicator, some magic bits, and highest 4 bits of the block address to port 0x1F6: outb(0x1F6, (addr >> 24) | 0xE0));
- Send the command (0x20) to port 0x1F7: outb(0x1F7,0x20).

Analisi del codice

La procedura bootmain è inclusa nel file /boot/main.c, e referenzia codice contenuto nei file inc/x86.h (per le procedure outb e insl) e nel file inc/elf.h per la definizione delle strutture Elf e Proghdr.

```
#include <inc/x86.h>
#include <inc/elf.h>
```

Grafo delle dipendenze di inclusione per main.c:



Le funzioni definite in /boot/main.c sono: `readsect, readseg, bootmain e waitdisk`, che sono chiamate con le seguente sequenza:

La prima funzione che analizziamo è la waitdisk la cui funzionalità è quella di verificare lo stato del controller, in particolare waitdisk verifica attraverso la porta 0x1F7 ed utilizzando un ciclo di busy waiting se il controller è READY. La comunicazione con il controller avviene utilizzando il busy waiting perché in questa fase gli interrupt sono disabilitati ed è quindi l'unico modo per poter interrogare le periferiche.

void waitdisk (void)

Definizione alla linea **99** del file **main.c**.

Questo è il grafo dei chiamanti di questa funzione:


```
void
waitdisk(void)
{
     while ((inb(0x1F7) & 0xC0) != 0x40)
          /* do nothing */;
}
```

Il valore esadecimale 0xC0 corrisponde in binario a 11000000 e affinchè il ciclo venga interrotto il valore della porta 0x1F7 messo in and logico con la maschera 0xC0 deve valere 01000000, che equivale ad affermare il il READY bit dello status bit del controller è on.

La funzione `readsect`

La funzione readsect legge un settore (512 byte) dal disco. La comunicazione con il controller avviene attraverso le porte da 0x1F0 a 0x1F7 usando le istruzioni outb e insl definite nel file (inc/x86.h) .

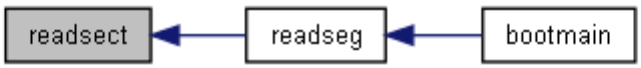
void readsect (void * dst,
uint32_t offset
)

Definizione alla linea **107** del file **main.c**.

Referenzia **SECTSIZE**, e **waitdisk()**.

Questo è il grafo delle chiamate per questa funzione:

Questo è il grafo dei chiamanti di questa funzione:




```
void
readsect(void *dst, uint32_t offset)
{
```

```
waitdisk();    //busy waiting sul controller

outb(0x1F2, 1);          // scrive sulla porta 0x1F2 il valore 1
outb(0x1F3, offset);
outb(0x1F4, offset >> 8);
outb(0x1F5, offset >> 16);
outb(0x1F6, (offset >> 24) | 0xE0); // Invia 0xE0 per HardDisk

outb(0x1F7, 0x20);  // comando 0x20 – lettura di un settore.
                    // Dopo questo comando il controller parte e si
                    // mette in busy. Quando ha letto tutto il
                    // settore si mette in ready.

waitdisk();

// read a sector

insl(0x1F0, dst, SECTSIZE/4); // Legge SECTSIZE/4 volte un long
dalla
                              // porta 0x1F0 e lo scrive in dst
                              // (memoria centrale)
}
```

La funzione readseg

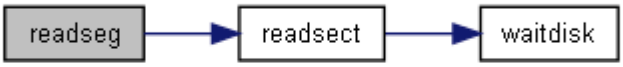
La funzione readsect vista precedentemete è usate per realzzare una funziobne di più alto livello, la readseg, il cui scopo è quello di trasferire in memoria centrale ad un indirizzo dato (I parametro), un certo numero di settore del disco (II parametro fornito alla procedura), a partire da un certo settore del disco (III parametro). Qui di seguito riportiamo la definizione della procedure readseg ed il relativo codice.

void readseg (uint32_t pa,
uint32_t count,
uint32_t offset
)

Definizione alla linea **72** del file **main.c**.

Referenzia **readsect()**, e **SECTSIZE**.

Questo è il grafo delle chiamate per questa funzione:



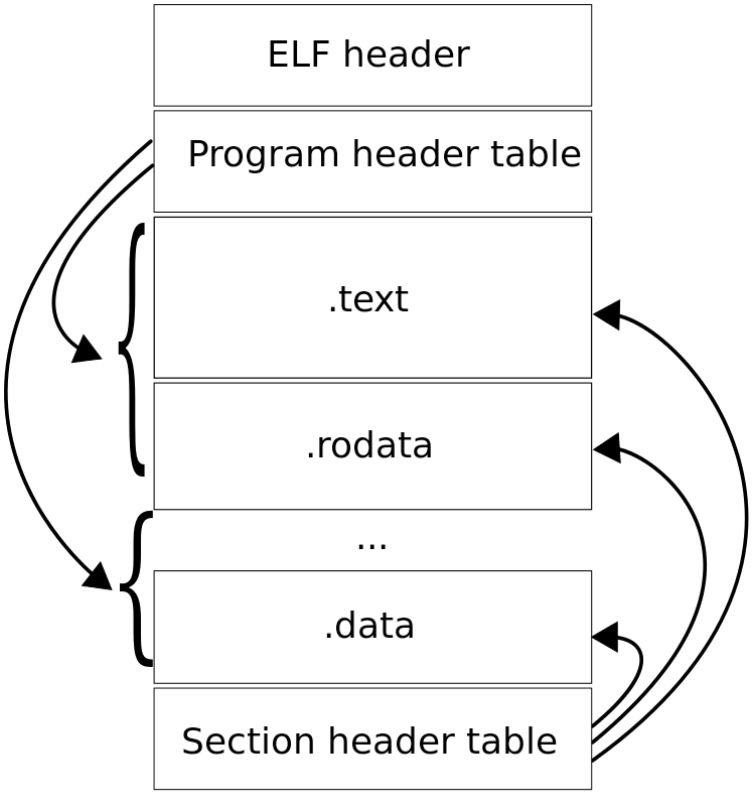
Questo è il grafo dei chiamanti di questa funzione:


```
#define SECTSIZE512
void
readseg(uint32_t pa, uint32_t count, uint32_t offset)
{
      uint32_t end_pa;

      end_pa = pa + count;

      // Allinea pa alla dimensione del blocco
      pa &= ~(SECTSIZE - 1);// ~(512 - 1) = ~(511) = ~(111111111) =
                            // 1111 1111 1111 1111 1111 1110 0000 0000
                            // pa &= 000000000

// Identifica il blocco dal quale iniziare a leggere considerando che il
byte
// 0 si trova nel settore 1, poichè il settore 0 contiene il bootloader

      offset = (offset / SECTSIZE) + 1;
      while (pa < end_pa)
      {     // ciclo per leggere tutti i settori richiesti
            readsect((uint8_t*) pa, offset);
            pa += SECTSIZE;
            offset++;
      }
}
```

Il caricamento del kernel

Il kernel è caricato su disco nel formato .elf. Si tratta di un formato di file standard comune per gli eseguibili, codice oggetto, librerie condivise e core dump pubblicato nella specifica System V Application Binary Interface, e poi nel Tool Interface

standard, è stato rapidamente accolto tra i diversi fornitori di sistemi Unix. Nel 1999 è stato scelto come formato di file binario standard per i sistemi Unix e Unix-like su x86. Un file in formato elf contiene tutte le informazioni necessarie al caricamento di un eseguibile in memoria centrale, ed è l'output di un processo di compilazione o linking. Un file di tipo ELF è composto da un header iniziale a lunghezza fissa che contiene un insieme di informazioni generali sul file, seguito da un program header a dimensione variabile che contiene informazioni sulle diverse sezioni/segmenti che compongono il programma. Il formato elf viene usato per rappresentare codice oggetto e codice eseguibile, nel primo caso diciamo che il programma è composta da sezioni, che saranno poi opportunamente accorpate per formare i segmenti nell'ambito di un eseguibile.



L'ELF header è così definito:

```
#define ELF_MAGIC 0x464C457FU /* "\x.ELF" in little */
struct Elf {
    uint32_t e_magic;    // must equal ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;
    uint16_t e_machine;
    uint32_t e_version;
    uint32_t e_entry;
    uint32_t e_phoff;
    uint32_t e_shoff;
```

```
    uint32_t e_flags;
    uint16_t e_ehsize;
    uint16_t e_phentsize;
    uint16_t e_phnum;
    uint16_t e_shentsize;
    uint16_t e_shnum;
    uint16_t e_shstrndx;
};
```

Questi i campi principali dell'elf heaer:

- e_entry: fornisce l'indirizzo virtuale delle prima istruzione eseguibile del programma a cui viene trasferito il controllo in fase di esecuzione. Se il file non un entry point associato a questo campo vale zero.
- e_phoff: contiene la posizione in termini di offset in byte rispetto all'inizio del file, del program header . Se il file non ha un program header , questo campo vale zero.
- e_ehsize: contiene le dimensioni in byte dell' ELF header.
- e_phentsize: contiene la dimensione in byte di un elemento del program header, tutti gli elementi hanno la stessa dimensione in byte,
- e_phnum: contiene il numero di elementi contenuti nel program header.

Questo invece il contenuto di un elemento del program header:

```
struct Proghdr {
     uint32_t p_type;
     uint32_t p_offset;
     uint32_t p_va;
     uint32_t p_pa;
     uint32_t p_filesz;
     uint32_t p_memsz;
     uint32_t p_flags;
     uint32_t p_align;
};
```

Riportiamo di seguito un esempio di section e program headers realtivi al kernel di JOS

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        f0100000 001000 004f35 00  AX  0   0 16
  [ 2] .rodata           PROGBITS        f0104f40 005f40 0016d0 00   A  0   0 32
  [ 3] .stab             PROGBITS        f0106610 007610 00882d 0c   A  4   0  4
  [ 4] .stabstr          STRTAB          f010ee3d 00fe3d 002c01 00   A  0   0  1
  [ 5] .data             PROGBITS        f0112000 013000 06be1b 00  WA  0   0 4096
  [ 6] .bss              NOBITS          f017de20 07ee1b 000ef0 00  WA  0   0 32
  [ 7] .comment          PROGBITS        00000000 07ee1b 000059 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 07ee74 00004c 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 07f078 000c60 10     10  65  4
  [10] .strtab           STRTAB          00000000 07fcd8 000af6 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0xf0100000 0x00100000 0x11a3e 0x11a3e R E 0x1000
  LOAD           0x013000 0xf0112000 0x00112000 0x6be1b 0x6cd10 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .stab .stabstr
   01     .data .bss
   02
```

Assumono particolare rilievo i campi VirtAddr ("VMA") e PhysAddr (LMA) o indirizzo di caricamento della sezione di testo. L'indirizzo di caricamento di una sezione definisce l'indirizzo fisico di memoria in cui tale sezione deve essere caricato in memoria. In un file di tipo ELF, questo dato è memorizzato nel campo p_pa del program header relativo alla sezione text.

Il campo VMA di una sezione (p_va) è invece l'indirizzo di memoria virtuale che viene assegnato al codice contenuto nella sezione. Quindi il linker genererà un codice eseguibile a partire dall'indirizzo p_va. Il mapping tra gli indirizzi virtuali e fisici sarà poi realizzato avvalendosi dei meccanismi di memoria virtuale.
Ad esempio, nel caso del sistema operativo Jos il Kernel del sistema operativo viene collocato ad un indirizzo virtuale molto alto, come 0xf0100000 (gli ultmi 256 MB si spazio in una memoria di 32 GB), in modo da lasciare la parte inferiore dello spazio di indirizzamento virtuale della memoria ai programmi utente.
Ovviamente, molti sistemi non hanno alcuna memoria fisica all'indirizzo 0xf0100000, quindi non possiamo contare sul fatto di poter memorizzare il kernel lì. Useremo quindi la gestione della memoria hardware del processore per mappare indirizzo 0xf0100000 virtuale all'indirizzo fisico 0x00100000 (dove il boot loader caricato il kernel in memoria fisica)

Analizziamo ora la funzione bootmain il cui scopo è leggere il file .elf che contiene il kernel (che ricordiamo in JOS risiede su HD a partire dal settore 1), estrarre dall'header le informazioni necessarie e con queste caricare il kernel in memoria centrale a partire dall'indirizzo 0x00100000.

void bootmain (void)

Definizione alla linea **39** del file **main.c**.

Referenzia **ELF_MAGIC**, **ELFHDR**, **Proghdr::p_memsz**, **Proghdr::p_offset**, **Proghdr::p_pa**, **readseg()**, e **SECTSIZE**.

Questo è il grafo delle chiamate per questa funzione:


```
void
bootmain(void)
{
    struct Proghdr *ph, *eph;     // inizializza due puntatori alle
                                  // strutture di elf e program header

    // leggi in memoria a partire dall'indirizzo 0x00100000 i primi 4KB
    // del kernel presente su disco a partire dal settore 1 (offset 0)

    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

    if (ELFHDR->e_magic != ELF_MAGIC)// verifica che il formato del
                                     // header sia corretto
            goto bad;

    // recupera tra i dati appena letti il program header e il numero di
    // elementi che contiene

    ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;

    // Carica in memoria tutti i segmenti  presenti nel kernel
    for (; ph < eph; ph++)
      readseg(ph->p_pa, ph->p_memsz, ph->p_offset);

// Cedi il controllo all'entry point

      ((void (*)(void)) (ELFHDR->e_entry))();

bad:
      outw(0x8A00, 0x8A00);
      outw(0x8A00, 0x8E00);
      while (1)
            /* do nothing */;
}
```