# Os from scratch

Di Silvio Abruzzo

Versione 2.0 (Riveduta da Ivan Gualandri)

## INDICE

# LEZIONE 0

## 1) Introduzione

Lo scopo di questo documento è quello di fornire le linee guida per la creazione di un sistema operativo (os) amatoriale. Tutti gli esempi sono basati sul mio "ItaliOs", un sistema operativo che ha la prerogativa di essere totalmente in italiano.
Lo scopo principale nel creare un os credo che sia il divertimento nel apprendere l'architettura del pc e nel programmare qualcosa di diverso dalla solita rubrica, o dal solito hello world scritto nel linguaggio di programmazione di turno.
Ovviamente per fare un os occorrono dei prerequisiti tutt'altro che banali: per seguire questa guida bisogna conoscere un minimo di c++, di asm e avere tanta fantasia. E' fondamentale avere delle basi teoriche, ad esempio avere letto qualche libro di teoria degli os: come quelli che si studiano nei corsi universitari. Cercherò comunque di essere il più chiaro possibile anche se probabilmente a volte darò per scontato cose che per chi non è del settore scontate non sono.
Ci divertiremo nel discutere tante soluzioni e a volte scegliere quella più semplice e veloce da implementare: infatti per cercare di fare qualcosa che non ci annoii spesso dovremo mettere da parte un po' di efficienza per non impantanarci in dettagli che ci farebbero passare la voglia di andare avanti. Naturalmente, si potrà sempre migliorare in seguito.
Mi riferirò al concetto espresso in questo capoverso come al "teorema della semplicità. Per leggere questa guida non è indispensabile avere il pc alla mano, infatti vi inserirò molto codice, renderò disponibile, inoltre, alla fine di ogni capitolo, tutto l'os da compilare e da modificare come spiegato nello stesso capitolo.

## 2) Come è fatto un OS

In questo capitolo cercheremo di dare una definizione di os, cercheremo di capire da quali parti è composto e poi cercheremo di definire le caratteristiche che deve avere. Anche se tutti noi abbiamo sempre a che fare con un sistema operativo, è molto difficile definirlo, perchè a seconda di come lo utilizziamo, soggetivamente, ne daremo diverse definizioni: una segretaria ad esempio ci dirà che un sistema operativo è quella cosa che gli permette di salvare i documenti e di lavorare con un word processor. Un bambino ci dirà che serve per farlo giocare, etc...
Possiamo definire un sistema operativo come una macchina virtuale e come un gestore di risorse. Una macchina virtuale perchè si occupa di definire un interfaccia unica e indipendente dall'hardware, in modo da sgravare il programmatore dal gestire il funzionamento dell'hardware. Gestore di risorse perchè si occupa di gestire le varie periferiche del nostro pc e ci permette un interazione con loro. Pertanto possiamo disegnare questo primo schema:


I due livelli sopra l'hardware sono l'os (kernel + applicazioni). Il kernel è formato da molte parti. Bisogna distinguere tra due tipi possibili di kernel: monolitico e microkernel. Esistono anche altri tipi di strutturazione del kernel, ma sono di fatto modelli teorici con scarsa applicabilità. I kernel monolitici al giorno d'oggi vengono un pò snobbati ma sono di fatto quelli più usati sopratutto nei progetti amatoriali.
Un kernel monolitico contiene al suo interno TUTTO quello che deve avere un kernel. Per definizione tutti i componenti sono molto legati tra loro. E' molto facile aggiungere driver, ma alla fine la pesantezza e la complessità renderanno impossibile ulteriori sviluppi. Una variante dei kernel monolitici sono i kernel modulari, che pur restando monolitici permettono di caricare moduli

dinamicamente. Questo meccanismo è molto ingegnoso perchè permette di conservare la facilità di sviluppo propria dei kernel monolitici, ma anche di mantenere efficente l'occupazione di memoria e di ridurre la complessità. I microkernel sono quelli più apprezzati perchè permettono di rendere facilmente espandibile, modificabile e personalizzabile il kernel. Il meccanismo si basa sul concetto che il kernel si occupa di gestire le operazioni fondamentali dell'os mentre la gestione di tutto il resto, memoria, net, fs, hardware, viene demandata a dei server che sono totalmente indipendenti dal kernel. I vantaggi sono evidenti, ma la realizzazione si complica sia per la gestione che per la lentezza dello switch tra i vari server. Cerchiamo ora di definire le caratteristiche che dovrà avere il nostro os:

- Essere facilmente portabile su altre piattaforme
- Mantenere un rapporto semplicità di sviluppo/efficenza tendente ad 1
- Poter funzionare sulla maggior parte dell'hardware esistente

Inizialmente ci concentreremo sullo sviluppo su piattaforme IA32, ma predisporremo in modo che sia possibile aggiungere altre piattaforme facilmente. La mia idea era quella di creare un piccolo kernell che configurasse tutti quegli aspetti dipendenti dalla piattaforma e che fornisse delle funzioni indipendenti usabili dal kernel vero e proprio. Ho poi chiamato questo kernellino col nome di ccs (sistema configurazione cpu). Il ccs viene caricato dal bootloader, si occupa di configurare tutti gli aspetti specifici della cpu e fornisce un interfaccia unica per il kernel vero e proprio che gestisce tutto l'hardware. Un esempio: il ccs fornisce una funzione per fare il task switch e poi il kernel fornisce le politiche di schedulazione che sono indipendenti dal processo che esegue il task switch e da come esso viene realizzato.


Definiremo in seguito con precisione le caratteristiche del ccs e del kernel. La prima cosa da fare ora e decidere qualche altro piccolo dettaglio, vedere gli strumenti che ci servono e iniziare a scrivere il bootloader

## 3) Gli Strumenti a disposizione

Il primo aspetto pratico da definire è il linguaggio da usare e quale compilatore usare.
Come linguaggio useremo il c++, perché ci permette di fare tutto quello che fa il c e inoltre ha delle caratteristiche interessati: mi riferisco all'overloading, ai parametri di default e ai namespace. A volte useremo anche meccanismi propri dell'OOP.
La scelta del compilatore ricade sul g++, compilatore c++ che fa parte dei tool del gcc. Fra le qualità ricordiamo:

- è open souce
- produce codice abbastanza pulito ed efficente.

Per fare in modo che il gcc mantenga codice non troppo dipendente da linux dobbiamo aggiungere alcune opzioni:

- *-fomit-frame-pointer* - serve a mantenere integrità negli indirizzi delle varie funzioni ed è obbligatorio
- *-Wall* - serve a dire al compilatore di mostrarci tutti i warning, si potrebbe utilizzare anche il

--pedantic, ma non non è consigliato perché da warning inutili sugli assegnamenti con indirizzi, sulla parte in asm e sulle direttive del gcc.

- *-march=i386* - indica che il nostro codice è per 386 e quindi non ottimizzera per processori superiori. Questa è necessaria perché se si utilizzano caratteristiche del processori superiori bisogna abilitarle ed al momento ciò non è ancora stato fatto.
- *-c* - indica che vogliamo solo la compilazione senza il linking
- *-nostdlib* - indica che non vogliamo che sia possibile usare la libreria standard, questo perché, ovviamente, la libc e la libc++ sono librerie dinamiche che lavorano in user mode con le api del sistema operativo.
- *-nostdinc* - fa in modo che non vengano usati gli header di default
- *-fno-builtin* - fa in modo che non vengano usati componenti interni che servono solo a linux
- *-fno-exceptions* - fa in modo che non usi le eccezioni perché esse sono dipendenti dall'os
- *-fno-rtti* - fa in modo che non possiamo usare la rtti per lo stesso motivo visto sopra

Alcune parti andranno scritte necessariamente in assembler e per queste verrà usato il nasm che è un assemblatore open source, funziona su molte piattaforme: useremo l'assembler SOLO nel ccs, altrimenti il kernel non sarà + multipiattaforma. Il gcc permette anche di inserire assembler inline purché sia in formato At&t. Possiamo inserire anche asm in formato intel, ma solo per le versioni del gcc superiori alla 3.0, noi ci limiteremo all'asm in formato at&t in modo da avere la maggiore compatibilità possibile.

La cosa più difficile per un os è il debug: noi useremo bochs che è un ottimo emulatore di pc e ha delle interessanti funzioni per il debug. Per compilarlo io vi consiglio di dare il ./configure con questi parametri:

```
./configure --enable-iodebug --enable-disasm --enable-debugger
```

Vediamo cosa significano questi argomenti:

- *--enable-debugger* - permette di abilitare il debugger di bochs che ha funzioni per il trace dei registri, controllo della memoria, etc
- *--enable-disasm* - permette di abilitare il disassembler di bochs: ci permette di vedere quali istruzioni vengono eseguite
- *--enable-iodebug* - permette di usare delle speciali porte di bochs per il debug.

Ora abbiamo tutti gli strumenti e le informazioni necessarie per iniziare: Vediamo nel prossimo capitolo come scrivere un bootloader.

# LEZIONE 1

## 1) Il Bootloader

Il nostro bootloader si chiama ItaLo (ItaliOs Loader), si occupa di leggere da floppy il ccs e il kernel, inizializza il pmode e quindi lancia il ccs, si tratta di un bootloader raw (cioè legge i dati direttamente da floppy, senza utilizzare nssun file system).
Appena un pc si avvia il bios compie le operazioni di bootstrap cioè controlla che tutto l'hardware sia funzionante, cerca di determinarne il tipo, la quantità di ram, i dischi disponibili, eccetera. Una volta compiute tutte queste operazioni il bios carica il primo settore del primo cilindro del dispositivo selezionato e inizia ad eseguire le istruzioni assembler presenti dall'indirizzo 0000:0x7C00.
Le caratteristiche obbligatorie del bootloader sono:

- non essere più grande di *512* byte
- terminare con *0xAA55*
- iniziare usando la modalità reale quindi niente registri a 32 bit e istruzioni del pmode (pmode = modalità protetta, la modalità a 32 bit dei processori intel)

Lo scopo del nostro bootloader è:

- caricare il ccs+kernel in memoria
- inizializzare la modalità protetta
- lanciare il ccs

### *Caricare il ccs+kernel*

Per caricare il kernel dobbiamo prima averlo scritto su un qualche dispositivo: per comodità useremo il floppy che è un dispositivo di facile programmazione, enorme diffusione, poco costo.
La prima cosa è decidere il formato di come saranno organizzate le informazioni, visto che non abbiamo ancora un file system. Un'organizzazione semplice può essere:

Come ci impone il Bios nel primo settore mettiamo il bootloader e di seguito mettiamo il ccs+kernel (la fine dipende dalla grandezza del kernel). In seguito vedremo che dobbiamo modificare leggermente l'impostazione del floppy.
Per fare una scrittura raw su linux possiamo usare dd. La sintassi di dd è:

```
dd if=immagine_del_floppy of=/dev/fd0
```

Una volta caricato il bootloader e il kernel su floppy, riavviando il pc, (per funzionare deve essere impostato per l'avvio da floppy prima degli altri device), il bootloader caricherà il kernel in memoria.
Per leggere da floppy ci viene in aiuto l'interrupt del bios numero 13h e precisamente il servizio 02h (Le informazioni sugli interrupt del bios sono prese dalla “lista di Ralf Brown” disponibile su: ).
Dunque per farlo funzionare dobbiamo impostare:

***INPUT:***
ah = 02h
al = numero di settori da leggere (non deve essere zero)
ch = 8 bit bassi del numero del cilindro
cl = numero del settore da 1 a 63 (bit 0-5)
2 bit alti del numero del ciclindro (bit 6-7)
dh = numero della testina
dl = numero del driver
es:bx = buffer dei dati
***OUTPUT:***

CF settato se c'è stato un errore
non settato se tutto è andato bene

Iniziamo a scrivere un pò di codice assembly.

```
        mov si,KRNL_SIZE ;lo usiamo come una specie di contatore
        mov bx,ax        ;metto ax a zero
        mov ax,0x1000  ;in ax  va l'offset in caricare il kernel
               ; per poi spostarlo in es
        mov es,ax      ;quindi setto es con il mio offset
        mov cx,2       ;cx = 2 per farlo partire dal 2° settore
        xor dx,dx
```

Il frammento di codice appena visto carica una serie di parametri per poter leggere i dati dal floppy. Proseguiamo:

```
.readAgain:
       mov ah,0x02      ;indico che voglio leggere
       mov al, 01       ;leggo solo un settore
       int 0x13         ;leggo dal floppy
       jc short .error  ;se c'è un errore prova la lettura 3 volte
       mov ax,es        ;
       add ax,32        ;       incrementento es di 32 byte
       mov es,ax        ;

       dec si           ;decremento il contatore
       jz short .readOk ;se si è diverso da 0 leggo il settore
                ;successivo
       inc cx           ;incremento il contatore dei settori
       cmp cl,18        ;controllo se è finita la traccia
       jbe short .readAgain  ;se non è ancora finita leggo il
                      ;settore successivo
        mov cl,1       ;altrimenti metto la traccia ad 1
        inc dh         ;e metto la seconda faccia
        cmp dh,2
        jne short .readAgain  ;se non sono alla seconda faccia
                       ;leggo ancora
        mov dh,0       ;altrimenti torno alla prima faccia
        inc ch     ;e incremento ancora la testina
        jmp short .readAgain  ;e vado a leggere

.error: ;se ci è stato un errore in lettura
        xor ax,ax     ;metto ax a zero
        int 0x16      ;aspetto la pressione di un tasto
        int 0x19      ;riavvio il caricamente del bootloader

.readOk:
        ; spengo il motore del floppy
        mov edx,0x3f2   ;metto edx=0x3f2 che è il comando per
               ;spegnere il floppy
        mov al,0x0c     ;metto al=0xC che è la porta del floppy
        out dx,al       ;eseguo il comando
```

La modalità reale per compatibilita con il buon vecchio 8086 non permette di accedere ad oltre 1 MB di ram, dobbiamo quindi attivare la A20 che è una linea del processore che risolve il problema. Le righe di codice per fare ciò sono queste:

```
        cli     ;disabililito gli interrupt
.wait1:        ;aspetto che la tastiera sia libera
        in   al, 0x64
```

```
        test al, 2
        jnz  .wait1
        mov  al, 0xD1
        out  0x64, al

.wait2:      ;aspetto che la tastiera sia disposta ad accettare comandi
        in   al, 0x64
        and  ax, byte 2
        jnz  .wait2
        mov  al, 0xDF
        out  0x60, al
```

Una volta caricato il kernel ed attivato l'A20 bisogna entrare in modalità protetta. I processori dal 286 in poi supportano una modalità in più rispetto ai processori precedenti: questa è la modalità protetta. Essa permette di trarre vantaggio delle caratteristiche a 32bit dal 386 in poi e delle caratteristiche a 24 bit del 286. Come abbiamo visto anche il 286 supporta la modalità protetta ma noi lavoreremo col 386 e superiori, sebbene le modifiche da fare siano minime.

Per entrare in modalità protetta prima bisogna caricare la gdt. La gdt (global description table) si occupa di gestire tutte le caratteristiche del pc, in una prossima lezione la esamineremo molto più attentamente. Una volta caricata la gdt basta attivare un bit nel registro cr0 e saltare ad un pezzo di codice a 32 bit, e il tutto si è compiuto.

```
        lgdt [gdtinfo]  ;carico la gdt

        mov eax,cr0   ;  metto in eat cr0
        or  al,1      ;imposto il bit per la modalità protetta
        mov cr0,eax   ;siiiiiiiii, vado in modalità protetta
        jmp dword (flat_code-gdt_table):pmode1
         ; tutto è pronto, configuro i selettori dei segmenti

pmode1:
[BITS 32]
```

Una volta arrivati in un pezzo di codice a 32 bit la prima cosa da fare è impostare i registri segmento e lo stack

```
    mov  ax, flat_data-gdt_table ;metto il selettore in ax
    mov  ds, ax                  ;uso uguale selettore per i dati
    mov  es, ax                  ;l'extra segment
    mov  fs, ax                  ; fs
    mov  gs, ax          ; gs
    mov  ss,ax                   ; e lo stack
    mov  esp,  0x9FFFC   ;imposto la base dello stack

    push dword 2            ;metto il valore 2 nello stack
    popfd                          ;imposto a 2 gli eflag
```

A questo punto finalmente si può saltare al codice del kernel che noi abbiamo caricato all'indirizzo 0x1000. Quindi basta fare:

```
     mov eax, 0x10000
        call eax
```

Forse risulta poco chiara la questione della gdt, ma tra qualche lezione verrà chiarita. I primi due punti della nostra scaletta sono stati affrontati, ora occupiamoci del ccs.

In questa lezione l'unica cosa che farà il kernel sarà quella di scrivere a video:

"Ciao sono il tuo sistema operativo".

Per fare questo dobbiamo andare a scrivere direttamente nella memoria della scheda video.
Gli indirizzi della scheda video sono mappati in memoria a partire da 0xb8000, ogni carattere a video è rappresentata da 2 byte quindi tutta la memoria video sarà grande 0xb8000+(80*25*2). 80 x 25 sono rispettivamente le colonne e le righe del nostro schermo.
I 2 byte rappresentano: il primo il codice ascii del carattere, il secondo il colore del carattere e sui attributi. Il secondo byte è organizzato così:

| **bit** | **funzione** |
|---|---|
| 0,1,2,3 | colore del carattere |
| 4,5,6 | colore dello sfondo del carattere |
| 7 | lampeggiamento |

Per ottenere un carattere bianco con sfondo nero basta impostare il tutto a 0x7.
Quindi scriviamo una semplice funzione che ci stampa a video un carattere.

```
char *videomem = (char *) 0xb8000;
void putc(char carattere){
videomem++ = carattere; /*imposto il carattere*/
videomem++ = 0x7;       /*imposto gli attributi del carattere*/
}
```

Ovviamente bisogna gestire gli accapo, lo scroll dello schermo, etc ma è una cosa che faremo nella prossima lezione. Bene! A questo punto abbiamo tutto. Bisogna fare alcune considerazioni finali. L'esecuzione del ccs non inizierà mai dal main ma dalla funzione _start. Quindi avremo una cosa del genere:

```
void _start(){
        main();
        while(1);
}

int main(){
        puts("Ciao sono il tuo sistema operativo");
        return 0;
}
```

Per questa lezione è tutto. Vi lascio allo studio dei sorgenti.
I file sono:

main.c che contiene il ccs
bootloader.asm che contiene il bootloader
makefile che contiene il makefile.

Il makefile ci permette:
make  compila il tutto
make install  compila e installa su floppy
make clean  cancella gli oggetti.

# LEZIONE 2

## 1) Canale a video

Per vedere quello che facciamo sulla macchina abbiamo bisogno di un output sullo schermo del monitor, che stampi scritte a video, che faccia scorrere il testo (scroll), che cancelli e pulisca lo schermo e stampi molti tipi base.
Per differenziarlo dalla cout del c++ standard noi lo chiameremo kout.
Vediamo innanzitutto di approfondire il funzionamento della scheda video. Di default lo schermo è impostato ad una risoluzione di 80x25 caratteri. Se vogliamo cambiare risoluzione del testo prima di andare in pmode nel bootloader dobbiamo inserire qualcosa del genere:

```
mov ax, 1112h
int 0x10
```

Queste 2 righe ci permettono di fare in modo che la risoluzione del testo sia 50x80. Date le mie scarse capacità visive useremo la 25x80 ^___^ .
Ogni carattere occupa in memoria 2 byte, il primo contiene il codice ascii del carattere, il secondo contiene gli attributi del carattere.


Mentre i colori sono questi:


I caratteri vanno copiati in memoria dall'indirizzo 0xB8000 che è proprio dove la scheda video cerca i dati per disegnare a video. Se vogliamo ad esempio scrivere "CIAO" su schermo partendo dall'angolo in alto a sinistra, con sfondo nero e scritta bianca, la memoria dovrà contenere una cosa del genere:

| | |
|---|---|
| 'C' | 0xB8000 |
| 7 | 0xB8001 |
| 'I' | 0xB8002 |
| 7 | 0xB8003 |
| 'A' | 0xB8004 |
| 7 | 0xB8005 |
| 'O' | 0xB8006 |
| 7 | 0xB8008 |

Quindi possiamo iniziare a scrivere:

```
enum  {
            NERO,
            BLU,
            VERDE,
            CELESTE,
            ROSSO,
            VIOLA,
            MARRONE,
            BIANCO_SPORCO,
            GRIGIO,
            BLU_CHIARO,
            VERDE_CHIARO,
            CELESTINO,
            ARANCIONE,
            ROSA,
            GIALLO,
            BIANCO,
            MAX_COLOR = 15,
            LAMPEGGIO = 128
};

typedef unsigned char colore;
```

Inizialmente la nostra classe può avere questo aspetto:

```
class Video{
public:
        Video();
        void clear() ;
        void put(const char c) ;
        void put(const char* c) ;
        void set_text_color( colore  mycolor);
        void set_back_color(colore mycolor);
        colore get_text_color(void);
        colore get_back_color(void);
private:
        word *videomem ; //puntatore alla memoria
        size_t off ;     //offset usato per le coordinate della y
        size_t pos ;     //indica la posizione della x
        //restituisce il colore nella forma color<<8
        word get_formed_color();
        colore color;
        //li metto come costati per dare un significato ai numeri
        static const size_t max_x = 80;
        static const size_t max_y = 25;
        static const size_t screen_size  = max_x * max_y;
};
```

L'unica cosa su cui soffermarci è il tipo della variabile videomem. Abbiamo usato una word perché questo ci permette di gestire ogni carattere come se fosse un singolo elemento.

Iniziamo dall'implementazione del costruttore:

```
        Video::Video(){
            pos=0 ;        off=0 ; //la posizione iniziale è 0,0
            //imposto il puntatore all'inizio della memoria video
          videomem = (word*) 0xB8000 ;
          set_text_color(GIALLO);
          set_back_color(BLU);
          clear(); //puliamo lo schermo
        }
```

Quindi vediamo subito la clear:

```
          void Video::clear()     {
                unsigned int i;
                for(i=0; i
```

La funzione get_formed_color() ci permette di avere il byte del colore organizzato nella maniera già vista sopra ma shiftato di 8 bit cosicché facilmente si aggiunge il codice ascci del carattere e si forma la word intera.
Vediamo le implementazioni quindi della get_formed_color(), della set_back_color() e della set_text_color():

```
word Video::get_formed_color(){
        return color<<8;
};

void Video::set_text_color( colore  mycolor){
        color = color & 0x70; //quindi ottengo 0xxx0000
        color |= (mycolor & 0xF);
}

void Video::set_back_color(colore mycolor){
        //azzero i bit deillo sfondo
        color &=0x8f; //ottengo x000 xxxx
        //preparo i bit dello sfondo
        mycolor &= 0x7;  //cancello tutti i bit tranne i primi tre quindi alla
fine ottengo 0000 0xxx
        mycolor <<=4;  //shifto tutto di 4 posizioni a sinistra in questo modo
li allineo per bene e ottengo 0xxx 0000
        //setto il colore
        color += mycolor;
}
```

I commenti mi sembrano abbastanza esplicativi, vediamo la funzione più importante di tutte, la put():

```
void Video::put(const char c){
      if(c == '\n'){
      accapo();
          return;
     }
     if(c == '\t'){
            for(int i = 0; i < tab_size; i++) put(' ');
          return;
     }
      if(pos>=max_x){
            accapo();
      }
      if(off>=screen_size){
      //faccio lo scroll di una riga dello schermo
            for(int i=0; i < screen_size - 80; i++)
                  videomem[i]=videomem[i+80];
            for(int  i = screen_size-80; i < screen_size; i++)
      videomem[i]=videomem[off+pos]=(unsigned char)' '|get_formed_color();
            pos = 0; off-=80;
     }
      videomem[off + pos] = (unsigned char) c | get_formed_color() ;
            pos++;
      }
```

Lo scroll dello schermo lo gestiamo copiando tutto il contenuto della memoria di una riga in alto.

La funzione che manda accapo risulta abbastanza semplice, basta semplicemente spostare il puntatore all'area di memoria alla riga successiva, questo si puo fare prendendo il valore dell'inizio della linea in cui ci troviamo ed aggiungendogli 2 la larghezza (ricordiamo che ogni carattere si rappresenta con 2 byte).
Bene il resto della classe lo si può vedere direttamente dal file che, tra l'altro, è ben commentato.

Analizziamo un aspetto legato alla nostra classe video. Dunque creiamo un oggetto globale di questo tipo:

```
Video mystdout;
Video& kout = mystdout;
```

Il nostro oggetto è globale; quindi all'avvio del css dobbiamo eseguire noi il costruttore, dato che non possiamo usare il loader degli eseguibili di linux.

## 2) Costruttori di oggetti globali

In questa lezione tratteremo sopratutto il linker ld, creeremo inoltre un bello script per il nostro ccs
Prima di tutto iniziamo a vedere cosa fa di preciso un linker. Un linker unisce oggetti e archivi di file, riloca i dati e il codice, sistema i simboli. Di solito il linkaggio è la fase seguente alla compilazione.
Per ora utilizziamo ld con questo comando:

```
ld  -Bstatic --oformat binary  -occs.bin -L./main --whole-archive
-t -lmain -Ttext 0x10000 -Map kernel.map
```

Con questo gli diciamo che il kernel deve essere totalmente statico, che il codice deve iniziare all'indirizzo 0x10000 e che ci deve produrre una mappa del kernel.
Dobbiamo creare il nostro script perché vogliamo conoscere a quali indirizzi il linker mette i costruttori e in questo modo poterli eseguire all'avvio del ccs.
Per non perderci nello studio di come sono fatti gli script di ld prendiamo il suo script di default e poi lo modifichiamo con le caratteristiche che ci interessano.
Per avere in output il file che viene usato bisogna usare l'opzione --verbose. Ecco il file opportunamente scritto, modificato per i nostri scopi.

```
/*File generato da LD per produrre binari standard e modificato per ottenere
caratteristiche particolari, come l'etichettamento delle zone di memoria dei
costruttori*/

OUTPUT_FORMAT("binary")
OUTPUT_ARCH(i386)
ENTRY(_start)
SECTIONS
{

  . = 0x10000;
   /*il codice*/
  .text           :  { *(.text .stub .text.* .gnu.linkonce.t.*)  }

  /*dati a sola lettura read only data*/
  .rodata         : { *(.rodata .rodata.* .gnu.linkonce.r.*) }
  . = DATA_SEGMENT_ALIGN(0x1000, 0x1000);
  . = ALIGN(32 / 8);
  .data           :
  {


  /*costruttori del c++*/
           __CTOR_LIST__ = .; LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
*(.ctors)   LONG(0) __CTOR_END__ = .;

         __DTOR_LIST__ = .; LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
*(.dtors) LONG(0) __DTOR_END__ = .;

    *(.data .data.* .gnu.linkonce.d.*)
    SORT(CONSTRUCTORS)
  }
  .ctors          :
```

```
  {
    KEEP (*crtbegin.o(.ctors))
    KEEP (*(EXCLUDE_FILE (*crtend.o ) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
  }
  .dtors          :
  {
    KEEP (*crtbegin.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o ) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
  }
  .bss            :  {
    *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
   . = ALIGN(32 / 8);
  }
  . = ALIGN(32 / 8);
  _end = .;
  PROVIDE (end = .);
  . = DATA_SEGMENT_END (.);
}
```

Le righe che ho aggiunto sono queste:

```
/*costruttori del c++*/
__CTOR_LIST__ = .; LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2) *(.ctors)
LONG(0) __CTOR_END__ = .;
__DTOR_LIST__ = .; LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2) *(.dtors)
LONG(0) __DTOR_END__ = .;
```

L'etichette __CTOR_LIST__ e __DTOR_LIST__ contengono l'inizio della zona di memoria ove ci sono gli indirizzi dei costruttori. Il primo valore in memoria contiene il numero dei costruttori. Quindi riusciamo a produrre questa funzione:

```
void __do_global_ctors (void)
{
    //la lista dei costruttori è definita nello scrpt di ld
  extern void (*__CTOR_LIST__)();
  void (**constructor)() = &__CTOR_LIST__;
  //il primo intero è il numero di costruttori
  int total = *(int *)constructor;
  constructor++;
  // eseguo i costruttori uno alla volta
  while(total--)
    {
      (*constructor)();
      constructor++;
    }
}
```

Ora è sufficiente fare in modo che questa funzione venga eseguita prima di caricare il main e abbiamo finito.

## 3) Uniamo il tutto

Bene ora dobbiamo unire il tutto per formare l'os.
Nella classe video abbiamo usato le funzioni itoa, utoa, etc. Io ho sviluppato un parte della libc, ma la includeremo in un secondo tempo nel kernel, per evitare di ingrandire il ccs inutilmente. Un cosa da ricordare è che si preferisce mettere più codice possibile in userland piuttosto che kerneland: parleremo ancora dei vari land in una prossima lezione.

Ho aggiunto quindi una cartella libc in include che contiene funzioni utili. Le funzioni le ho messe inline e non in un altro file sperando che il gcc le includa direttamente nel codice, dato che sono abbastanza piccole. Ho aggiunto anche stddef.h mettendo le definizioni di size_t , ptrdiff_t, NULL e alcuni tipi utili:

```
typedef unsigned char byte;
typedef unsigned short int word;
typedef unsigned int dword;
typedef unsigned long long qword; //8 byte
```

Modifichiamo anche il main.c facendo in modo che invece di usare la vecchia funzione usi la nostra kout.
Vi lascio allo studio dei sorgenti.

# LEZIONE 3

## 1) Bochs

La cosa più difficile in un os, come avevo già accennato, è il debug: uno strumento utilissimo a tale scopo è bochs, un emulatore di pc che ci offre delle caratteristiche particolarmente utili.
Innanzitutto scarichiamo i sorgenti di bochs da http://bochs.sourceforge.net Scompattiamo il tarball e diamo un bel:
./configure --enable-iodebug --enable-disasm --enable-debugger

- *--enable-debugger* permette di abilitare il debugger di bochs che ha funzioni per il trace dei registri, controllo della memoria, etc
- *--enable-disasm* permette di abilitare il disassembler di bochs, ci permetterà di vedere quali istruzioni vengono eseguite
- *--enable-iodebug* ci permette di usare delle speciali porte di bochs per il debug.

Quindi diamo un bel make (e volendo anche un bel make install), e installiamo i font di bochs:

```
cp font/vga.pcf font-path-here/misc
compress font-path-here/misc/vga.pcf
mkfontdir font-path-here/misc
xset fp rehash
```

Sostituiamo font-path-here con il path di dove abbiamo i font. Solitamente è:
/usr/X11R6/lib/X11/fonts/ . Quindi riavviamo xfs.
Io personalmete uso questo file di configurazione: bochsrc. Per ulteriori informazioni consiglio la lettura della documentazione di bochs.
Può essere utile durante l'esecuzione far stampare dei messaggi di debug che non devono comparire, invece, quando eseguiamo l'os su un pc reale. Per far stampare un carattere nel logger di bochs basta mettere il codice ascii nella porta 0xe9 e questo come per magia apparirà nel prompt di bochs.
Ho creato una classe debug del tutto identica alla kout. Col tempo la modificheremo per renderla molto più potente.

Altre cose interessanti che ci fornisce bochs sono: il trace dei registri, il trace delle istruzioni e il blocco per poi eseguire il codice passo a passo.
I comandi vanno messi nella porta 0x8A00 : eccone una breve lista:

- 0x8A00: Usato per abilitare il debug
- 0x8A01: Selects register 0: Indirizzo dell'inizio del monitoraggio della memoria (inclusivo)
- 0x8A02: Selects register 1: Indirizzo della fine del monitoraggio della memoria (esclusivo)
- 0x8A80: Abilita il monitoraggio della memoria come indicato dai registri 0 e 1 e cancella entrambi i registri
- 0x8AE0: Ritorna il Prompt di debug
- 0x8AE2: Disabilita il trace delle istruzioni
- 0x8AE3: Abilita il trace delle istruzioni
- 0x8AE4: Disabilita il trace dei registri
- 0x8AE5: Abilita il trace dei registri (funziona solo se il trace delle istruzioni e abilitato)
- 0x8AFF: Disabilita l'i/o debug e le funzioni di monitoraggio della memoria.

Bene! Ora possiamo implementare tante belle funzioni che ci saranno molto utili. Il debugger lo attiveremo direttamente da costruttutore in questo modo non ci dovremo preoccupare di sapere se è attivato oppure no. L'unica cosa che ci rimane da progettare è la maniera di attivare i vari trace. Ci si prospettano varie soluzioni:

```
void istruction_trace(bool enabled);
```

oppure

```
void istruction_trace(word status);
```

o ancora

```
        void set_istruction_trace();
        void unset_istruction_trace();
```

A mio parere la migliore è la seconda perché a differenza della prima ci permette di definire ulteriori stati di abilitato o disabilitato in caso di espansione futura. La terza invece è la peggiore perché dobbiamo implemenare delle funzioni in più, con lo svantaggio aggiuntivo di dover ricordare più nomi.
Ecco il codice dell'header e della classe.
ccs/include/debug.h:

```
#ifndef _DEBUG_H_
#define _DEBUG_H_

#include
#ifndef endl
#define  endl  '\n'
#endif

enum{
        ON, OFF
};

class Debug{
public:
        Debug();
        void put(const char c) ;
        void put(const char* c) ;
        Debug& operator << (const int numero);
        Debug& operator << (const unsigned int numero);
        Debug& operator << (const char* str);
        Debug& operator << (const char ch);
        void istruction_trace(word status);
        void register_trace(word status);
        void stop();
};

extern Debug debug;

#endif
```

ccs/main/debug.cc:

```
#include < debug.h >
#include < string.h >
#include < io.h >

namespace{
        static const word BOCHS_IODEBUG_PORT = 0x8A00;
}

Debug debug;

void Debug::istruction_trace(word status){
        if(status == ON){
                outportw(BOCHS_IODEBUG_PORT, 0x8AE3);
                return;
        }
        if(status == OFF){
                outportw(BOCHS_IODEBUG_PORT, 0x8AE2);
                return;
        }
        return;
```

```
}


void Debug::register_trace(word status){
        if(status == ON){
                istruction_trace(ON);
                outportw(BOCHS_IODEBUG_PORT, 0x8AE5);
                return;
        }
        if(status == OFF){
                istruction_trace(OFF);
                outportw(BOCHS_IODEBUG_PORT, 0x8AE4);
                return;
        }
        return;
}

void Debug::stop(){
        outportw(BOCHS_IODEBUG_PORT, 0x8AE0);
}

Debug& Debug::operator << (const int numero){
        char num_str[30];
        itoa(numero, num_str);
        put(num_str);
        return *this;
}


Debug& Debug::operator << (const unsigned int numero){
        char num_str[30];
        utoa(numero, num_str, 10);
        put(num_str);
        return *this;
}


Debug& Debug::operator << (const char* str){
        put(str);
        return *this;
}

Debug& Debug::operator << (const char ch){
        put(ch);
        return *this;
}

Debug::Debug(){
        outportw(BOCHS_IODEBUG_PORT, 0x8A00);
        put("\n\nDebugger inizializzato\n\n");
}

void Debug::put(const char* c){
        int i;
        for (i = 0; c[i] != '\0'; i++) {
                put(c[i]);
        }
        return;
}

void Debug::put(const char c){
        outportb(0xe9, c);
}
```

Anche questa volta abbiamo finito, vi lascio allo studio dei sorgenti: ho aggiunto un nuovo file io.h con delle funzioni per i/o inoltre ho modificato la kout per convogliare tutto quello che stampa a video nel logger di bochs.

# LEZIONE 4
## 1) Uno Sguardo panoramico ad IA32

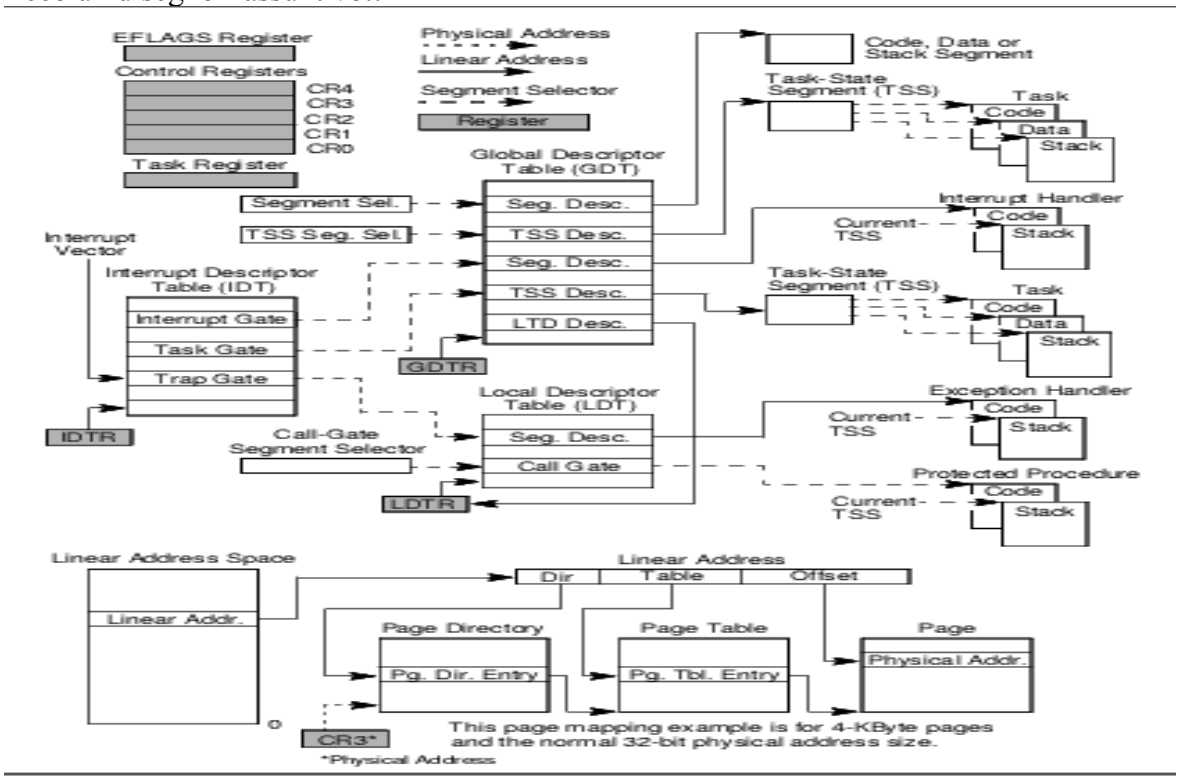
### *L'architettura del processore*
Affrontiamo ora degli argomenti di notevole impegno. Il nostro scopo per la fine di questa lezione è:

1. Configurare i segmenti di memoria
2. Configurare gli interrupt e gli irq

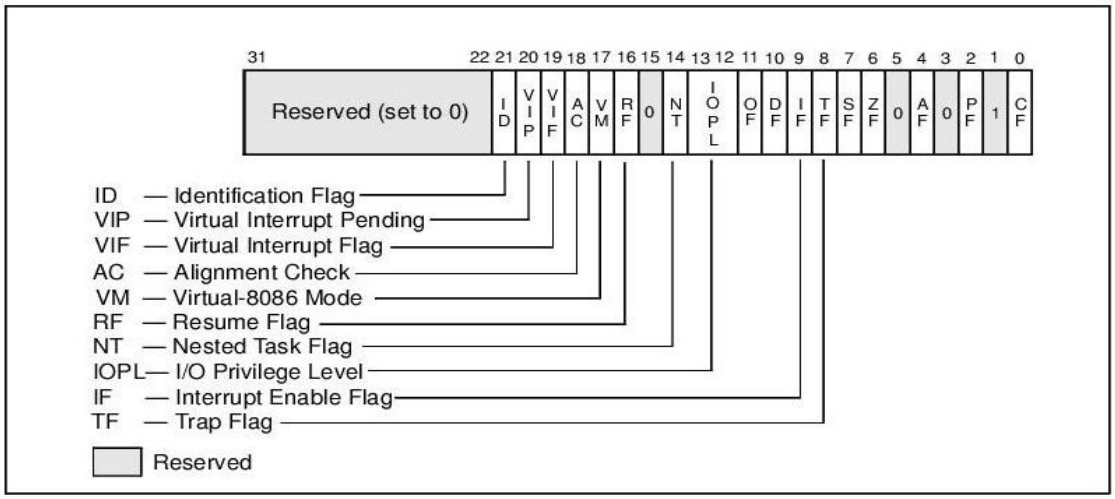
Per iniziare impariamo a conoscere come funziona l'architettura IA32. Per ulteriori approfondimenti leggere i tre manuali della intel, sopratutto il terzo. I disegni sono stati presi da li. Da buon despota illuminato spesso cercherò di rendere più facile il tutto tralasciando di dire alcune cose, oppure semplificando i concetti. Innanzitutto analizziamo come funziona IA32 e che caratteristiche ci offre. Un processore IA32 sostanzialmente ha 3 modi di funzionamento (in realtà sono 4) la modalità reale, la modalità protetta e la modalità virtual 8086.
Appena viene avviato il pc, il processore è in modalità reale e si comporta a tutti gli effetti come un 8086 ma molto piu veloce.
La modalità protetta è la modalità che è stata introdotta col 286 (e poi espansa col 386+) e permette di sfruttare tutte le potenzialità dell'architettura IA32. Infine abbiamo la modalità virtual 8086 che ci permette di avere un ambiente in modalità reale ma dentro la modalità protetta.
Noi ci concentreremo sulla modalità protetta nei processori da 386 in su.
A partire dal 386 abbiamo dei registri in più e delle strutture che ci permettono di configurare la memoria, i processi, etc. I registri general purpuse sono stati espansi a 32 bit (eax, ebx, ecx, edx, esi, edi) anche lo stack pointer e il base pointer sono diventati a 32 bit (esp, ebp); anche l'istruction pointer (eip) e i flag (eflag) sono stati estesi. Inoltre abbiamo dei registri nuovi:

- GS e FS sono registri segmento senza una specifica funzione
- cr0 --> cr4 sono 5 nuovi registri di controllo per processore
- gdtr che contiene la descrizione della gdt
- idtr che contiene la descrizione della idt
- ldtr che contiene la descrizione della ldt
- tr che contiene l'indirizzo del primo tss
- gestione della paginazione.

Ecco un disegno riassuntivo::



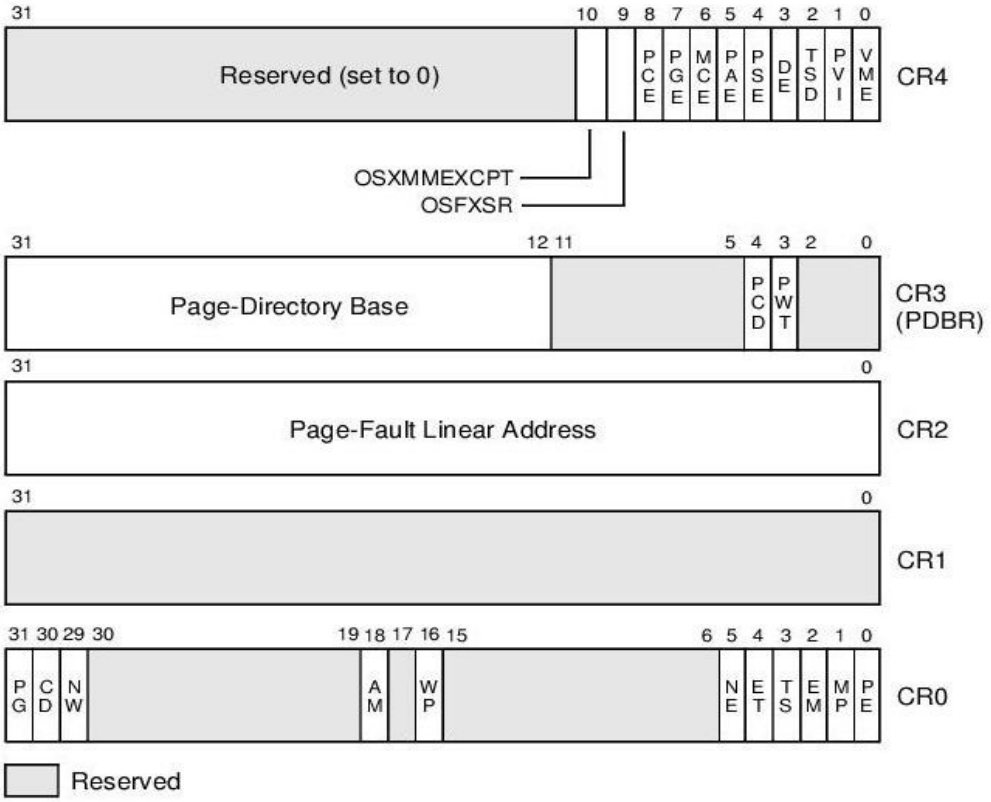
Iniziamo ad analizzare gli eflag:




TF: Da settare per abilitare la modalità single-step per il debug, cancellarlo per disabilitare questa modalità. In single-step model il processore genera un eccezione di debug dopo ogni istruzione. Se un applicazione setta il TF flag usando POPF, POPFD o IRET, l'eccezione viene generata dopo questa istruzione.

- IF: indica se gli interrupt sono abilitati
- VM: Virtual-8086 mode: da settare per abilitare l'emulazione dell'8086.
- ID: Indica la capacità di usare l'istruzione cpuid

Gli alri flag non ci interessano o comunque sono banali: basta vedere nel manuale della intel. Analizziamo i registri di controllo:




Iniziamo con cr0:

- PE: Protection enable. Abilita la modalità protetta quando settato e abilita la modalità reale quando è spento. Quando si abilita il pmode automaticamente non viene anche attivata la paginazione.
- WP: Quando abilitato non permette la scrittura di pagine a sola lettura a livello utente da parte dei supervisori (CPL < 3)mentre quando è disabilitato permette di scriverci

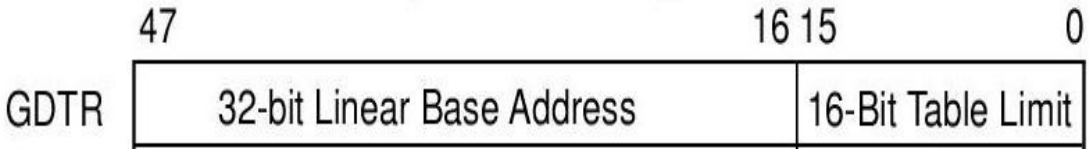
sempre per CPL < 3.
- PG: Quando è acceso permette e abilita la paginazione. Per avere effetto deve essere attivo anche PE.

Useremo intensivamente cr2 e cr3 quando programmeremo la paginazione, quindi ce ne occuperemo un altra volta.

**2) La tabella dei descrittori globali**

Tabella dei descrittori globali

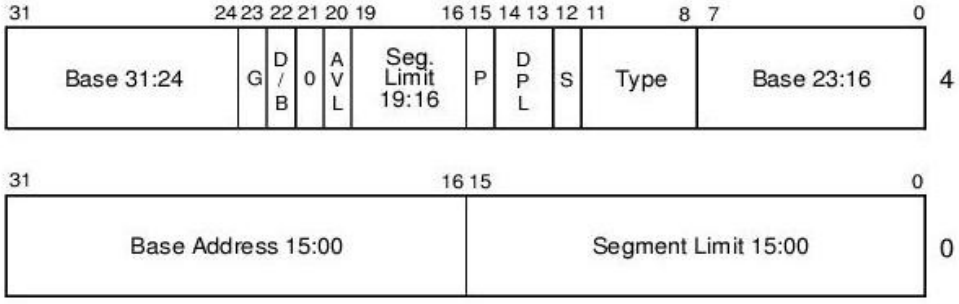
Una struttura fondamentale di IA32 è la gdt (tabella dei descrittori globali). La gdt è sostanzialmente un vettore grande al massimo 8192 elementi. Ogni elemento è un descrittore di segmento, grande 8 byte. Il primo elemento di questo vettore, cioè quello con indice 0 deve valere 0.
Per indicare la gdt al processore dobbiamo salvare la grandezza della gdt e l'indirizzo di partenza del registro gdtr. Il gdtr è fatto così:




Una semplice funzione per impostare il gdtr è questa:

```
void set_gdtr(qword * base,size_t num_desc) {
        dword gdt_reg[2];
        gdt_reg[0]=(num_desc*8) << 16;
        gdt_reg[1]=(dword)base;
        __asm__ __volatile__ ("lgdt (%0)": :"g" ((char *)gdt_reg+2));
}
```

Vediamo un po' i tipi di segmenti che esistono.
La forma generale di un descrittore di segmento è questa:
Dunque un descrittore di segmento indica il range di memoria a cui si riferisce e poi specifica gli attributi per questa zona di memoria. Poi vedremo come indicare al processore quale segmento




AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

usare. Ecco una descrizione dei campi:

- *Base*: E' l'indirizzo fisico (o lineare se è attivata la paginazione) nell'inizio del segmento di memoria
- *Limite*: Indica l'offset di quanto è grande un segmento:
- Se il flag G è spento allora ogni unita è di 1 byte, ad esempio un limite di 100

significa che il nostro segmento va da base a base+100 byte. Quindi possiamo descrivere segmenti che vanno da 1 byte a 1 MegaByte

- Se il flag G è acceso allora ogni unità è di 4KBytes (4096 byte), quindi ad esempio un limite di 100 significa che il nostro segmento va da base a base+(4096*100) byte. Quindi possiamo descrivere segmenti che vanno da 0 a 4 GigaByte.
- *Tipo*: Indica il tipo si segmento che si vuole costruire. Ecco delle tabelle dei tipi:
- Se S non è settato allora parliamo di un descrittore non di memoria:

| Type Field | | | | | Description |
|---|---|---|---|---|---|
| Decimal | 11 | 10 | 9 | 8 | |
| 0 | 0 | 0 | 0 | 0 | Reserved |
| 1 | 0 | 0 | 0 | 1 | 16-Bit TSS (Available) |
| 2 | 0 | 0 | 1 | 0 | LDT |
| 3 | 0 | 0 | 1 | 1 | 16-Bit TSS (Busy) |
| 4 | 0 | 1 | 0 | 0 | 16-Bit Call Gate |
| 5 | 0 | 1 | 0 | 1 | Task Gate |
| 6 | 0 | 1 | 1 | 0 | 16-Bit Interrupt Gate |
| 7 | 0 | 1 | 1 | 1 | 16-Bit Trap Gate |
| 8 | 1 | 0 | 0 | 0 | Reserved |
| 9 | 1 | 0 | 0 | 1 | 32-Bit TSS (Available) |
| 10 | 1 | 0 | 1 | 0 | Reserved |
| 11 | 1 | 0 | 1 | 1 | 32-Bit TSS (Busy) |
| 12 | 1 | 1 | 0 | 0 | 32-Bit Call Gate |
| 13 | 1 | 1 | 0 | 1 | Reserved |
| 14 | 1 | 1 | 1 | 0 | 32-Bit Interrupt Gate |
| 15 | 1 | 1 | 1 | 1 | 32-Bit Trap Gate |

- Se S è settato allora parliamo di un descrittore di memoria di codice o dati:

| Type Field | | | | | Descriptor Type | Description |
|---|---|---|---|---|---|---|
| Decimal | 11 | 10 E | 9 W | 8 A | | |
| 0 | 0 | 0 | 0 | 0 | Data | Read-Only |
| 1 | 0 | 0 | 0 | 1 | Data | Read-Only, accessed |
| 2 | 0 | 0 | 1 | 0 | Data | Read/Write |
| 3 | 0 | 0 | 1 | 1 | Data | Read/Write, accessed |
| 4 | 0 | 1 | 0 | 0 | Data | Read-Only, expand-down |
| 5 | 0 | 1 | 0 | 1 | Data | Read-Only, expand-down, accessed |
| 6 | 0 | 1 | 1 | 0 | Data | Read/Write, expand-down |
| 7 | 0 | 1 | 1 | 1 | Data | Read/Write, expand-down, accessed |
| | | C | R | A | | |
| 8 | 1 | 0 | 0 | 0 | Code | Execute-Only |
| 9 | 1 | 0 | 0 | 1 | Code | Execute-Only, accessed |
| 10 | 1 | 0 | 1 | 0 | Code | Execute/Read |
| 11 | 1 | 0 | 1 | 1 | Code | Execute/Read, accessed |
| 12 | 1 | 1 | 0 | 0 | Code | Execute-Only, conforming |
| 13 | 1 | 1 | 0 | 1 | Code | Execute-Only, conforming, accessed |
| 14 | 1 | 1 | 1 | 0 | Code | Execute/Read-Only, conforming |
| 15 | 1 | 1 | 1 | 1 | Code | Execute/Read-Only, conforming, accessed |

- DPL: livello di privilegio del descrittore. Dunque IA32, quando si è in pmode, prevede 4 livelli di protezione chiamati ring (anelli). Il più privilegiato è il ring 0 (livello del kernel), mentre il meno privilegiato è il ring 3 (livello utente). Noi useremo solo il 3 e lo 0.
- P: presente. Indica se un segmento è presente oppure no. Se si tenta si impostare un segmento non presente viene generata una bella #NP (segment not present).

- D/B: è consigliabile per i segmenti a 32 bit che sia ad 1, e per i segmenti a 16 bit che sia 0. Ha un sacco di funzioni in base al tipo di segmento. Per ulteriori informazioni vedere il manuale della intel a pagina 3-12.

Un ultima cosa da precisare è come indicare al processore i descrittori da usare. Dobbiamo rammentare che il processore ha degli specifici registri segmento. Per la precisione: CS, DS, SS, ES, FS, GS. I più importanti da impostare sono i primi 3. Il primo descrive il segmento per il codice, il secondo il segmento per i dati e il terzo il segmento per lo stack. Basta inserire l'indice del segmento nella gdt del segmento che ci interessa in uno di questi registri per fare in modo che la cpu li usi. L'eccezione è rappresentata da cs. Per indicare al processore il nuovo valore di cs, basta fare un jump al segmento in questione.
Ora possiamo iniziare ad implementare quello che abbiamo visto. Ci sono varie soluzioni per i nostri scopi: possiamo implementare tutto il meccanismo completo definendo decine di costanti per ogni tipo di descrittore, quindi creare varie funzioni per utilizzare tutta questa roba, oppure possiamo creare un becero vettore di qword, impostare come servono a noi i vari descrittori mettendone direttamente i valori e commentando adeguatamente il tutto. Per il teorema della facilità, visto che una volta impostati i descrittori non verranno più modificati, scegliamo quest'ultima soluzione.
Per semplicità creiamo una bella funzione del tipo:

```
void init_gdt();
```

e ne mettiamo il prototipo in gdt.h.

Visto che poi per implementare la protezione della memoria implementeremo la paginazione, creiamo ora 2 segmenti che prendono tutta la memoria, da 0 a 4 Gb, un segmento sarà per il codice, l'altro per i dati.

```
void init_gdt(){
        gdt[0] = 0;
        //Limite = 0xFFFFF (4 gb)
        //Base= 0
        //Ring= 0
        //Granularità = si
        //Tipo= 0xA  (codice con permesso di esecuzione e lettura)
        //Presente= si
        gdt[1] = 0x00CF9A000000FFFFLL;


        //Limite = 0xFFFFF (4 gb)
        //Base= 0
        //Ring= 0
        //Granularità = si
        //Tipo= 2  (Dati con permesso di lettura e scrittura)
        //Presente= si
        gdt[2] = 0x00CF92000000FFFFLL;

        set_gdtr(gdt, 3);
        //setto gli eflag o meglio li azzero tutti

        debug << "sto per settare gli eflag e i registri segmento" << endl;
        debug.register_trace(ON);

        asm("pushl $2; popf");
        //l'indice del segmento dei dati è 2, quindi la sua distanza dall'inizio
è 0x10 (16 = 8 * 2)
        //mentre l'indice del segmento dei dati è 1 quindi la sua distanza è 8
        asm volatile (
                "movw $0x10,%ax     \n"
                "movw %ax,%ds      \n"
                "movw %ax,%es      \n"
                "movw %ax,%fs      \n"
```

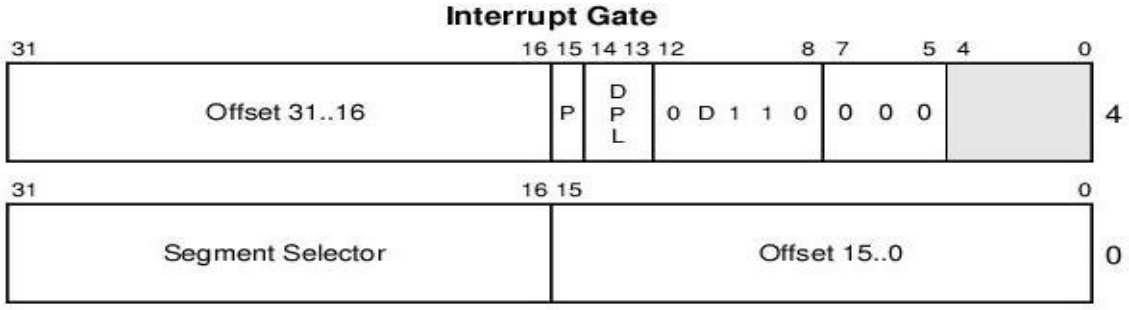
```
                "movw %ax,%gs      \n"
                "movw %ax,%ss      \n"
                "ljmp $0x08,$next    \n"
                "nop\n"
                "nop\n"
                "next:\n");

        debug.register_trace(OFF);
        return;
}
```

Molto bene! Per il momento abbiamo finito la gdt, ora passiamo alla idt

## 2) La tabella dei descrittori globali

Dopo aver implementato i segmenti dobbiamo capire come funziona in pmode la gestione degli interrupt e implementarla.
Quando entriamo in pmode tutti gli interrupt del bios che possiamo usare e che abbiamo usato tranquillamente in real mode non sono + disponibili. Il processore cerca gli handler degli interrupt in un vettore di descrittori del tutto simile alla gdt. Questo vettore può essere grande 256 elementi (o meno) e il suo indirizzo va memorizzato nell'idtr che è un registro identico (nella sua composizione) all'gdtr.
Il descrittore degli interrupt è fatto così:




- segmento selector: qui dobbiamo inserire l'offeset dall'inizio della gdt del segmento che contiene l'handler del nostro interrupt (nel nostro caso 0x8)
- offset: è l'eip del nostro handler
- D: dimensione, se ad 1 indica che il nostro segmento è a 32 bit altrimenti è a 16 bit
- P: presente, se ad 1 indica che il nostro interrupt è presente e viceversa
- DPL: indica il livello di privilegio

Detto ciò possiamo implementare facilmente tutta la gestione degli interrupt. Questa sarà leggermente + complessa dell'implementazione della gdt perkè gli interrupt sono cose che si modificano facilmente e quindi ci serve una maggiore flessibilità.
Questa è l'interfaccia per accedere e modificare gli elementi della idt:

```
namespace idt{
        //inizializza la idt
        void init();
        //installa l'handler func per l'interrupt index
        bool add(void (*func)(), size_t index);
        //cancella l'handler di un interrupt e restituisce il puntatore al
precedente handler
        void* del(size_t index);
        //restituisce l'indirizzo di un handler
        void* get(size_t index);
};
```

Ecco le implementazioni della add e della init:

```
namespace {
        //come è strutturato un int_gate
        struct int_gate{
                word offset_low;
                word segment;
                word reserved : 5;
                word option : 11;
                word offset_high;
        };

        //handler di default: non fa nulla
        void mint_null(){    asm("iret");      }

        //numero di interrupt che usiamo (il massimo!!!)
        static const size_t _num_elem = 256;

        //il nostro bellissimo vettore di interrupt ^__^
        int_gate  _IDT[_num_elem];

        //funzione per impostare l'idtr
        void set_idtr(int_gate* base,size_t num_desc) {
                dword idt_reg[2];
                idt_reg[0]=(num_desc*8) << 16;
                idt_reg[1]=(dword)base;
                __asm__ __volatile__ ("lidt (%0)": :"g" ((char *)idt_reg+2));
        }
}
bool idt::add(void (*func)(), size_t index){
        if(index < _num_elem){  //controlliamo se è un interrupt valido
                debug << "Imposto l'handler " << index << endl;
                dword addr = (dword)func;
                _IDT[index].offset_low = (addr & 0xFFFF);       //prendiamo i 16
bit + bassi
                _IDT[index].offset_high = (addr >> 16); //prendiamo i 16 bit +
alti
                _IDT[index].segment = 0x8;      //questo è il segmento codice
                _IDT[index].option = 0x470;   //100  0111  0000 (PRESENTE
DPL=0)
                return true;
        }
        debug << "Impossibile imposta  l'handler " << index << " si trova fuori
dal range. Massimo 256" << endl;
        return false;
}

void idt::init(){
        debug << "Inizializzazione degli interrupt in corso....." << endl;
        //impostiamo tutti gli interrupt con l'handler di default
        for(size_t i = 0; i < _num_elem; i++)
                add(mint_null, i);
        //settiamo l'idtr
        set_idtr(_IDT, _num_elem);

        //attiviamo gli interrupt
        asm("sti");
}
```

Il codice è supercommentato quindi dovrebbe essere abbastanza facile da capire. Esistono degli interrupt speciali. Questi sono le eccezioni e gli irq. Dovremo gestire entrambe le categoria in modo particolare.

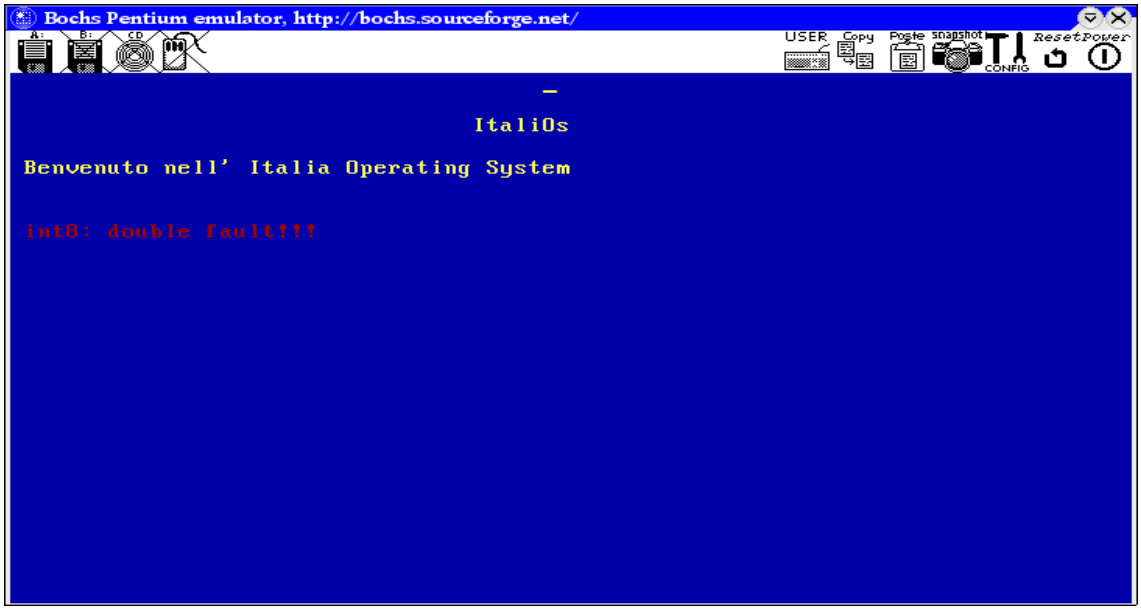
Ecco una tabella delle eccezioni:
Saggiamente ci occupiamo di gestire queste eccezioni. Creiamo un handler per ciascusa eccezione

| Vector No. | Mnemonic | Description | Type | Error Code | Source |
|---|---|---|---|---|---|
| 0 | #DE | Divide Error | Fault | No | DIV and IDIV instructions. |
| 1 | #DB | Debug | Fault/ Trap | No | Any code or data reference or the INT 1 instruction. |
| 2 | — | NMI Interrupt | Interrupt | No | Nonmaskable external interrupt. |
| 3 | #BP | Breakpoint | Trap | No | INT 3 instruction. |
| 4 | #OF | Overflow | Trap | No | INTO instruction. |
| 5 | #BR | BOUND Range Exceeded | Fault | No | BOUND instruction. |
| 6 | #UD | Invalid Opcode (Undefined Opcode) | Fault | No | UD2 instruction or reserved opcode.[1] |
| 7 | #NM | Device Not Available (No Math Coprocessor) | Fault | No | Floating-point or WAIT/FWAIT instruction. |
| 8 | #DF | Double Fault | Abort | Yes (Zero) | Any instruction that can generate an exception, an NMI, or an INTR. |
| 9 | | Coprocessor Segment Overrun (reserved) | Fault | No | Floating-point instruction.[2] |
| 10 | #TS | Invalid TSS | Fault | Yes | Task switch or TSS access. |
| 11 | #NP | Segment Not Present | Fault | Yes | Loading segment registers or accessing system segments. |
| 12 | #SS | Stack-Segment Fault | Fault | Yes | Stack operations and SS register loads. |
| 13 | #GP | General Protection | Fault | Yes | Any memory reference and other protection checks. |
| 14 | #PF | Page Fault | Fault | Yes | Any memory reference. |
| 15 | — | (Intel reserved. Do not use.) | | No | |
| 16 | #MF | Floating-Point Error (Math Fault) | Fault | No | Floating-point or WAIT/FWAIT instruction. |
| 17 | #AC | Alignment Check | Fault | Yes (Zero) | Any data reference in memory.[3] |
| 18 | #MC | Machine Check | Abort | No | Error codes (if any) and source are model dependent.[4] |
| 19 | #XF | Streaming SIMD Extensions | Fault | No | SIMD floating-point instructions[5] |
| 20-31 | — | Intel reserved. Do not use. | | | |
| 32-255 | — | User Defined (Nonreserved) Interrupts | Interrupt | | External interrupt or INT *n* instruction. |

che è fatto così:

```
void intnum(){
        kout << "\nIntnum: descizione";
        while(1);
}
```

Nelle prossime lezioni ci occuperemo di gestire accuratamente alcune eccezioni. Proviamo ora ad eseguire questo codice e ci viene restituita a video una cosa del genere:
Questo succede perché bisogna gestire gli irq.



**4) Interrupt Request**

Gli irq vengono gestiti dal chip 8259. Ogni chip riesce a gestire fino a 8 irq, vengono messi 2 chip in cascata per poter gestire fino a 16 irq. Ecco qui una tabella che riassume le funzioni corrispondenti ai vari irq:

| | |
|---|---|
| 0 | Timer |
| 1 | Tastiera |
| 2 | cascata |
| 3 | COM2/COM4 |
| 4 | COM1/COM3 |
| 5 | LPT2 |
| 6 | floppy |
| 7 | LPT1 |
| 8 | Real time Clock |
| 9 | libero |
| 10 | libero |
| 11 | libero |
| 12 | libero |
| 13 | coprocessore |
| 14 | primo hard disk |
| 15 | secondo hard disk |

I due chip dell'Irq vengono comandati tramite due tipi di comandi, ICW e OCW che vanno inviati sequenzialmente:

- ICW: Word per il comando di inizializzazione
- OCW: Word per il comando di operazione

Ecco una descrizione dei vari comandi:
ICW1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | LTIM | 0 | SNGL | ICW4 |

Allora:

- ICW4: indica se spediamo il comando ICW4 oppure no (per gli IA* da mettere SEMPRE ad 1)
- SNGL:indica quanti pic abbiamo
- quando a 0 indica 2 PIC in cascata
- quando a 1 indica solo il PIC master
- LTIM:indica come interpretare gli interrupt
- quando 0 indica di lanciare INTR sul fronte di salita
- quando a 1 indica di lanciare INTR al cambiamento di livello

ICW2

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| off7 | off6 | off5 | off4 | off3 | 0 | 0 | 0 |

Qui dobbiamo inserire il numero dell'interrupt che rappresenza IRQ0.

ICW3 Indica a quale IRQ è collegato il canale slave sul canale master. Di solito viene collegato sul canale IRQ2.

ICW4

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | SFNM | BUF | M/S | AEOI | CPU |

- CPU: modello di cpu usato
- 1 = modo IA*
- 0 = modo MC-80/85
- AEOI
- 1 = EOI automatico
- 0 = EOI manuale
- M/S
- 1 = master PIC
- 0 = slave PIC
- BUF:
- 1 = modo bufferizzato
- 0 = modo non bufferizzato
- SFNM:
- 1 = irq innestati
- 0 = irq non innestati

Ora che conosciamo i comandi possiamo fare una funzione che ci inizializza l'8259. Per comunicare con l'8259 si usano le porta 0x20 e 0x21 per l'8259 principale e 0xA0 e 0xA1 per il controller slave.

```
enum{
        M_PIC =  0x20,      // I/O for master PIC
        S_PIC =  0xA0   , // I/O for slave PIC
        M_VEC =  0x20 ,    //Dove viene mappato IRQ0
        S_VEC =  M_VEC+8   ,//Dove viene mappato IRQ8
        ICW1 =  0x11 ,   // 0001   0001
        ICW2_M = M_VEC,                         // ICW2 per il master
        ICW2_S = S_VEC    ,                     // ICW2 per lo slave
        ICW3_M =  (1 << 2),             // ICW3 per il master
        ICW3_S = 3                ,        // ICW3 per lo slave
        ICW4 =  0x01, //ICW4
};

//Inizializza l'8259
void Init8259(void){
        outportb_p(M_PIC, ICW1);        // Inizio l'inizializzazione
        outportb_p(S_PIC, ICW1);

        outportb_p(M_PIC + 1, ICW2_M);  //la base nel vettore di interrupt
        outportb_p(S_PIC + 1, ICW2_S);

        outportb_p(M_PIC + 1, ICW3_M);  //la cascata
        outportb_p(S_PIC + 1, ICW3_S);

        outportb_p(M_PIC + 1, ICW4);    //finisco l'inizializzazione
        outportb_p(S_PIC + 1, ICW4);
}
```

Spero che i commenti al codice siano chirificatori. Per finire con l'8259 dobbiamo vedere come esso gestisce l'abilitazione o la disablitazione di un irq.
L'8259 contiene tre registri:

1. IRR = interrupt request register, in questo registro sono memorizzati tutti gli irq che sono in attesa di essere serviti
2. ISR = interrupt service register, indica quale irq in questo momento si sta eseguendo
3. IMR = interrupt masked register, indica quel irq deve essere preso in considerazione dall'8259

Pertanto dobbiamo impostare l'IMR di ognuno dei due 8259 per specificare l'interrupt che vogliamo abilitare o disabilitare. Quindi possiamo scrivere queste 2 funzioni:
word irq_mask;

```
/*abilita la ricezioni di un irq*/
bool enable_irq(word irq_no){
        //controllo se è un irq valido
        if(irq_no <= 15){
                //faccio in modo che il bit corrispondente all'irq valga 0.
                //Es. se irq_no vale 6 succede ciò:
                //(1 << irq_no)   =  1000000
                //~(1 << irq_no) = 0111111
                //irq_mask &= ~(1 << irq_no);  =  xxxx xxxx x0xx xxxx
                irq_mask &= ~(1 << irq_no);
                 //se l'irq è maggiore di 8 mi assicuro che la linea 2 (quella
della cascata) sia attivata
                if (irq_no >= 8)
                        irq_mask &= ~(1 << 2);

                //sendo all'8259 la nuova configurazione dell'IMR
                outportb_p(M_PIC + 1, irq_mask & 0xFF);
                outportb_p(S_PIC + 1, (irq_mask >> 8) & 0xFF);
                return true;
        }
        return false;
}

/* disabilita la ricezione di un irq */
bool disable_irq(word irq_no){
        if(irq_no <= 15){
                  //mi occupo di mettere ad 1 il bit corrisponde all'irq che ci
interessa
                //es. se irq_no vale 3
                //(1 << irq_no)  =  1000
                //irq_mask |= (1 << irq_no)  =  xxxx xxxx xxxx 1xxx
                irq_mask |= (1 << irq_no);

                  //se gli irq alti (del pic slave) lo disabilito così è un po'
più efficente ^__^
                if ((irq_mask & 0xFF00) == 0xFF00)
                        irq_mask |= (1 << 2);

                outportb_p(M_PIC + 1, irq_mask & 0xFF);
                outportb_p(S_PIC + 1, (irq_mask >> 8) & 0xFF);
                return true;
        }
        return false;
}
```

Fino a qui dovrebbe essere tutto chiaro, ci manca solo un ultima funzione da implementare ed è esattamente quella per capire quale irq è in esecuzione effettivamente, per fare ciò basta esaminare il registro ISR.

```
size_t get_cur_irq(){
        byte irq;

        //indico che voglio leggere l'ISR
        outportb_p(M_PIC, 0x0B);
        irq = inportb_p(M_PIC);

        //se l'8259 master non ha nessun interrupt attivo controllo lo slave
        if (irq == 0) {
                outportb_p(S_PIC, 0x0B);
```

```
                irq = inportb_p(S_PIC);
               return find_first_bit(&irq, sizeof(byte)) ;
        }
        return find_first_bit(&irq, sizeof(byte));
}
```
La funzione find_first_bit cerca il primo bit settato, essa è definita nell'header bitops.h. Finisce qui questa lezione sull'irq.

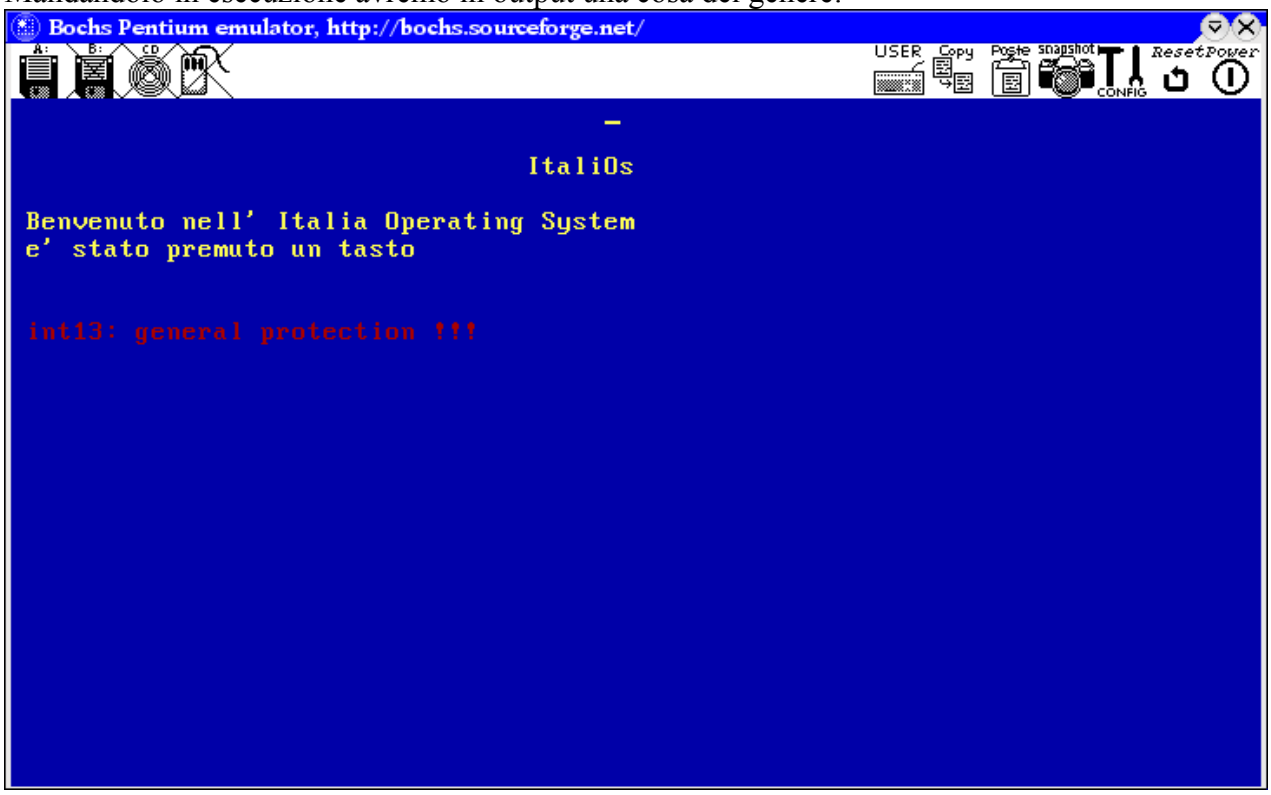
**6) Proviamo gli IRQ**

Ora cerchiamo di fare un piccolo programmino per provare se tutto il marchigegno implementato per gli irq e gli interrupt funziona.
Quando si preme un tasto viene generato l'irq numero 1. Useremo quindi la tastiera per provare gli irq!!!
L'handler è semplicemente questo:

```
void tastiera_handler(){
        //svuotiamo il buffer della tastiera
        inportb(0x60);
        kout << "è stato premuto un tasto" << endl;
        //diamo l'eoi
        outportb(0x20, 0x20);

        //ritorniamo dall'interrupt
        asm("iret");
}
```
e nel main aggiungiamo una cosa del genere:

```
        idt::add(tastiera_handler, M_VEC+1);
        enable_irq(1);
```
Mandandolo in esecuzione avremo in output una cosa del genere:

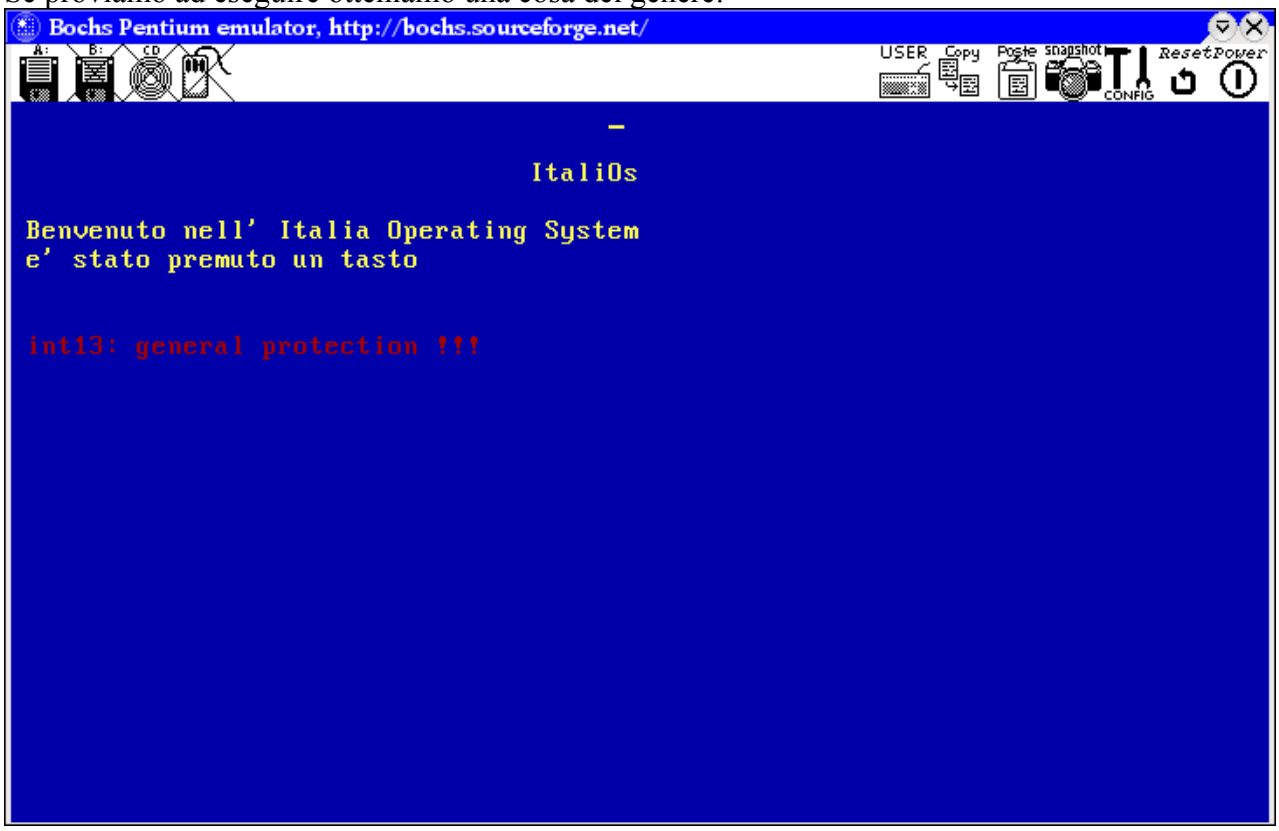



Ad un attenta analisi vediamo che c'è un errore che facciamo: non salviamo i registri all'inizio dell'handler e alla fine dell'handler. quindi lo modifichiamo così:

```
void tastiera_handler(){
        //salviamo i registri
        asm("pusha");
        //svuotiamo il buffer della tastiera
        inportb(0x60);
```

```
        kout << "è stato premuto un tasto" << endl;
        //diamo l'eoi
        outportb(0x20, 0x20);
        //ripristiniamo i registri
        asm("popa");
        //ritorniamo dall'interrupt
        asm("iret");
}
```

Se proviamo ad eseguire otteniamo una cosa del genere:




Come si può vedere abbiamo ottenuto ben poco. La causa di ciò è da attribuire al modo nel quale il gcc trasforma una funzione in assembly. Proviamo a disassemblare l'handler e vedere cosa ci ha prodotto. Per fare in modo che il compilatore ci faccia vedere il codice asm dobbiamo usare l'opzione -S

```
.globl _Z16tastiera_handlerv
        .type   _Z16tastiera_handlerv,@function
_Z16tastiera_handlerv:
        subl    $20, %esp
#APP
        pusha   /*salviamo tutti i registri*/
#NO_APP
        movl    $96, %edx
#APP
        inb %dx,%al     /*svuotiamo il buffer della tastiera*/
#NO_APP
        pushl   $10
        subl    $12, %esp
        pushl   $.LC3
        pushl   kout
        call    _ZN5VideolsEPKc /*stampiamo il messaggio*/
        addl    $20, %esp
        pushl   %eax
        call    _ZN5VideolsEc   /*stampiamo l'endl*/
        movl    $32, %edx
        movb    $32, %al
```

```
#APP
        outb %al,%dx    /*diamo eoi*/
        popa    /*ripristiniamo i registri*/
        iret    /*usciamo dall'interrupt*/
#NO_APP
        addl    $28, %esp
        ret
```

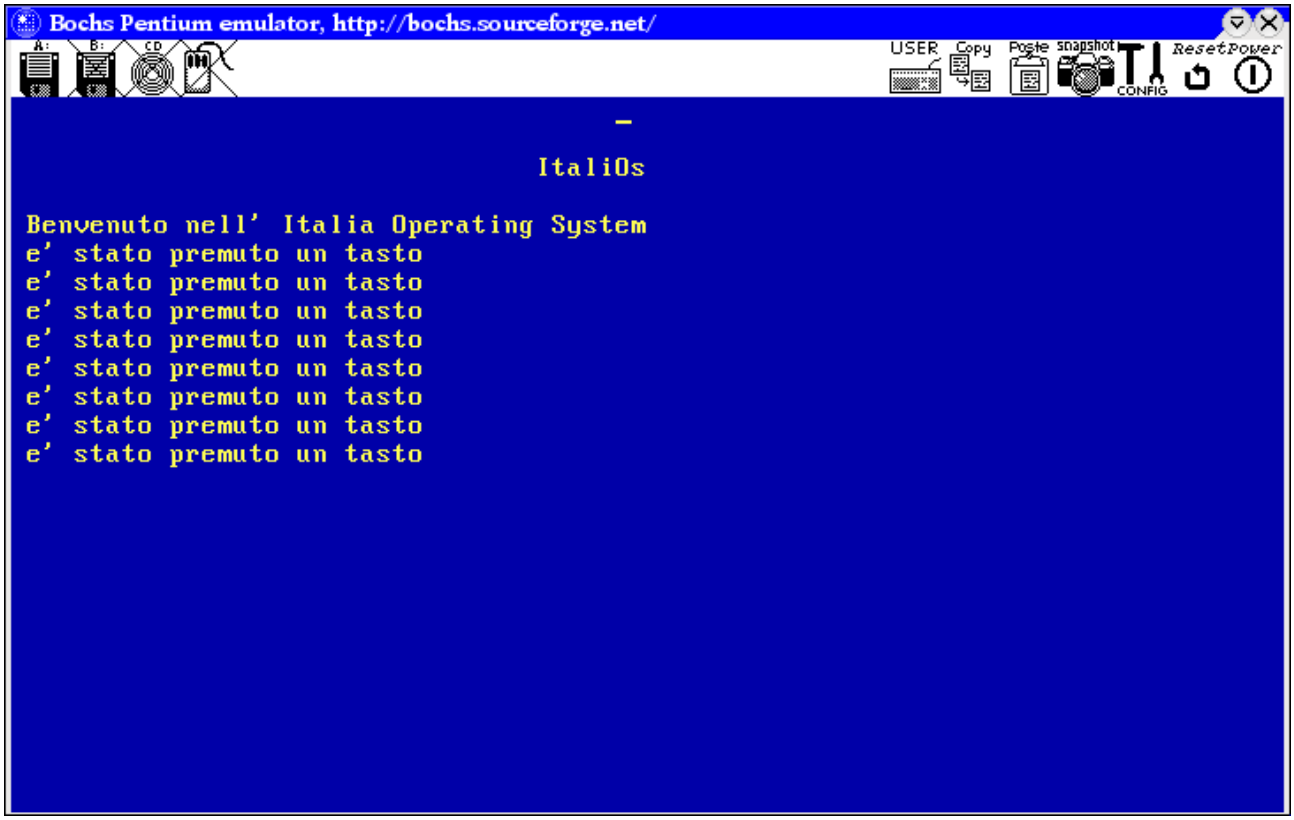
Vediamo che prima ancora che noi diamo la pusha il gcc modifica il valore dello stack, in questo modo la popa metterà dei valori sbagliati nei registri e la iret compirà il passo conclusivo per il #GP. Per risolvere questo problema bisogna implementare l'handler in asm e poi da questo chiamare la funzione vera e propria. Dobbiamo fare così:

```
        asm(".globl internal_handler        \n"
                "internal_handler:              \n"
                "pusha  \n"
                "call _Z16tastiera_handlerv   \n"
                "movl   $32, %edx       \n"
                "movb   $32, %al        \n"
                "outb %al,%dx    /*diamo eoi*/  \n "
                "popa   \n"
                "iret                \n"        /* Exit interrupt
*/
                );
```

e modifichiamo l'handler così:

```
void tastiera_handler(){
      //svuotiamo il buffer della tastiera
        inportb(0x60);
        kout << "e' stato premuto un tasto" << endl;
}
```

Mhhh, non preoccupiamo del nome schifoso con cui abbiamo usato la call, dipende da come il gcc decodifica i nomi delle funzioni del c++ (presto risolveremo il problema). A questo punto funziona tutto a dovere.




E' abbastanza chiaro che bisogna implementare un modo semplice di usare gli irq, che non ci faccia preoccupare di tutti questi dettagli ogni volta. Il modo più semplice è usare una funzione e poi richiamare l'handler giusto. Ecco l'implementazione.

```
//il tutto è dichiaro come extern C per non avere problemi nel chiamare le varie
variabili o funzioni
extern "C"{

        //vettore per i 16 interrupt
        void (*handler[16])();

        //handler di default
        extern void internal_handler(void);

        //quando nessun handler è impostato per quell'irq
        void null_handler(){ return;}

        //si occupa di dare l'eoi
        void free_irq(size_t number){
                if(number < 8)
                        outportb(m_pic, eoi);
                else
                        outportb(s_pic, eoi);
                return;
        }


        void  _do_irq(){
                //determino quel'è l'irq che si sta servendo
                byte current_irq = get_cur_irq();

                //quindi lo eseguo
                (*handler[current_irq])();

                //dò l'eoi
                free_irq(current_irq);
                return;
        }


        asm(".globl internal_handler        \n"
                "internal_handler:              \n"
                "cli  /*disabilito gli interrupt*/      \n    "
                "pusha          /*salvo tutti i registri*/   \n      "
                "call _do_irq    /*eseguo l'handler vero e proprio*/  \n"
                "popa       /*ripristino i registri*/   \n"
                "sti          /*riabilito gli interrupt*/    \n"
                "iret     /* esco dagli interrupt       */           \n"
                );


        /*funzione per aggiungere un irq*/
        void add_irq(void (*func)(), byte number){
                handler[number]=func;
                return;
        }
}       //fine dell'extern

/*questa funzione è dinuovo a regime di c++*/
void init_irq(){
                Init8259();     //inizializzo l'8259
                //imposto i vari handler
                for(int i = 0; i < 16; i++){
                        if(i < 8)
                                idt::add(internal_handler, M_VEC+i);
                        else
                                idt::add(internal_handler, S_VEC+i);
                        add_irq(null_handler, i);
```

```
                }
                return;
}
```
Bene! Arrivati a questo punto la gestione degli irq è diventata banale e ci si può concentrare solo sull'handler. Tutto il codice è nei file irq.cc e irq.h. Anche per questa volta abbiamo finito. In questa lezione abbiamo implementato molte cose, dalla gdt agli irq passando per la idt. La prossima volta implementeremo un gestore della memoria che sfrutti il paging.