

Esercizi con l'assemblatore NASM

Corso di Sistemi Operativi @ Unimi

Mattia Monga

6 marzo 2019

Contents

1	Programmare senza il sistema operativo	1
1.1	Programma finale	2
2	Somma	2
2.1	Il programma finale	3
3	Somma con syscall	4
3.1	Programma finale	5
4	Somma con scanf e printf	5
4.1	Il programma finale	6

1 Programmare senza il sistema operativo

La macchina all'accensione funziona in **modalità reale** (in pratica in una modalità che emula i processori degli anni '80): a 16 bit e senza separazione di privilegi. Tutte le risorse hardware sono accessibili direttamente al programmatore. La memoria è segmentata (ma non paginata).

`bits 16`

Il programma viene caricato dal firmware a un indirizzo prefissato (0000:7C00).

`org 0x7C00`

Il programma (lungo esattamente 512 byte) viene caricato solo se termina con la parola (2 byte) 0xAA55.

```
times 510-($-$$) db 0 ; 0-padding
dw 0xAA55
```

Per scrivere sullo schermo è sufficiente assegnare valori appropriati agli indirizzi che mappano la scheda video: si tratta di $80 \times 25 = 2000$ coppie di byte a partire dall'indirizzo B800:0000. Il primo byte codifica (in ASCII) il carattere da mostrare, il secondo le sue caratteristiche visuali.

```

mov ax, 0xb800      ; text video memory
mov ds, ax         ; ds non accessibile direttamente

```

Usando il registro bx come indice:

```

mov byte [ds:bx], 'm' ; indirizzamento relativo a ds
mov byte [ds:bx+1], 0x0F ; attrib = white on black

```

1.1 Programma finale

```

bits 16
org 0x7C00

mov ax, 0xb800      ; text video memory
mov ds, ax         ; ds non accessibile direttamente
start:
  mov bx, 10
write:
  cmp bx, 0
  jz end

  mov byte [ds:bx], 'm' ; indirizzamento relativo a ds
  mov byte [ds:bx+1], 0x0F ; attrib = white on black

  sub bx, 2
  jmp write
end:
  hlt

times 510-($-$$) db 0 ; 0-padding
dw 0xAA55 ; (boot_sig)

```

Il programma va compilato in formato binario (pura sequenza di codice macchina)

```
nasm -f bin mioboot-nobios-simple.nasm -o mioboot-nobios-simple.bin
```

Con QEmu a questo punto si può usare il file ottenuto come se fosse il **boot sector** del primo hard disk.

```
qemu-system-i386 -hda mioboot-nobios-simple.bin
```

2 Somma

Passiamo ora a una macchina dotata di sistema operativo e di due livelli di privilegio. Il sistema operativo esegue le proprie istruzioni in **kernel mode** (ring 00 su i386), le applicazioni in **user mode** (ring 11 su i386). S'immagini ora di voler scrivere un programma utente che fa la somma di 42 e 24. Essenzialmente:

```

mov eax, 42
add eax, 24

```

Per poterlo eseguire occorre utilizzare le astrazioni fornite dal sistema operativo e rispettare un certo numero di convenzioni che il sistema operativo impone.

In un sistema Linux i programmi generalmente adottano il formato ELF (**Executable and Linkable Format**). Ci deve essere un simbolo `main` che è il punto di partenza dell'esecuzione.

```
global main
```

Le istruzioni stanno poi nel segmento `.text`.

```
segment .text  
main:
```

Le variabili in memoria (anziché nei registri) si conservano nel segmento `.data`.

```
segment .data
```

Per esempio una variabile intera x (**Double word**: 2 parole, 4 byte) inizializzata a 0.

```
x: dd 0 ; dd è una pseudo istruzione NASM il cui mnemonico sta per Data Double (word)
```

Il risultato della somma lo salviamo in x .

```
mov eax, 42  
add eax, 24  
mov [x], eax
```

2.1 Il programma finale

```
global main  
segment .text  
main:
```

```
mov eax, 42  
add eax, 24  
mov [x], eax
```

```
segment .data
```

```
x: dd 0 ; dd è una pseudo istruzione NASM il cui mnemonico sta per Data Double (word)
```

Il programma va compilato in formato ELF.

```
nasm -f elf somma.nasm -o somma.o
```

Va anche **collegato** (**link editing**), anche se in questo caso non vengono collegate librerie ma viene solo adattato lo spazio di indirizzamento secondo le convenzioni del sistema operativo.

```

# il link editor sarebbe ld, ma l'utilizzo tramite gcc è più semplice
gcc -o somma somma.o
# Eventualmente, su una macchina a 64bit
# gcc -m32 -o somma somma.o

```

A questo punto però per vedere il risultato della somma bisogna eseguire il programma monitorandone l'esecuzione con il **debugger**

```

gdb somma
# break main
# run
# stepi
# stepi
# stepi
# print/d x

```

3 Somma con syscall

Volendo mostrare sullo schermo il risultato della somma, facendo quindi uso di una risorsa hardware, c'è bisogno di appoggiarsi al sistema operativo, tramite una **syscall**. La syscall **write** viene attivata caricando 4 nel registro **EAX** e sollevando l'interruzione software convenzionale (**0x80**). In **EBX** va indicata la destinazione della scrittura (un **file** speciale chiamato **stdout** e associato al numero 1), in **ECX** l'indirizzo della stringa da scrivere e in **EDX** la lunghezza della stringa (non si dipende, come invece in C, da terminatori convenzionali).

```

mov eax, 4           ; syscall 4 (write)
mov ebx, 1          ; file descriptor (stdout)
mov ecx, msg        ; stringa
mov edx, msg_size   ; dimensione stringa
int 0x80

```

Stampare il risultato della somma richiederebbe computazione aggiuntiva: il numero (66) va trasformato nella stringa di caratteri ASCII con cui ci aspettiamo di leggerlo ("66" corrispondente ai due byte 0x3636). Per semplicità stampiamo una stringa fissa.

```

; è buona abitudine terminare le stringhe con il carattere ASCII zero
msg: db 'Ciao!',10,0 ; 10 è il carattere 'a capo'
msg_size equ $ - msg ; pseudo istruzione (macro); $ è l'indirizzo corrente

```

Essendo dati in sola lettura, le stringhe generalmente vengono conservate nel segmento **.rodata**.

```

segment .rodata

```

Stampando 66 si ottiene sullo schermo il carattere 'B' (66 nella codifica ASCII).

```

mov eax, 42
add eax, 24

```

```

mov [x], eax

mov eax, 4 ; syscall 4 (write)
mov ebx, 1 ; file descriptor (stdout)
mov ecx, x ; l'indirizzo di partenza
mov edx, 1 ; 1 byte
int 0x80

```

Al termine del programma è anche opportuno fare una syscall `exit` (identificatore 1) per finirne l'esecuzione ordinatamente.

```

mov eax, 1 ; syscall 1 (exit)
int 0x80

```

3.1 Programma finale

```

global main
segment .text
main:

mov eax, 4 ; syscall 4 (write)
mov ebx, 1 ; file descriptor (stdout)
mov ecx, msg ; stringa
mov edx, msg_size ; dimensione stringa
int 0x80

mov eax, 42
add eax, 24
mov [x], eax

mov eax, 4 ; syscall 4 (write)
mov ebx, 1 ; file descriptor (stdout)
mov ecx, x ; l'indirizzo di partenza
mov edx, 1 ; 1 byte
int 0x80

mov eax, 1 ; syscall 1 (exit)
int 0x80

segment .data
x: dd 0 ; dd è una pseudo istruzione NASM il cui mnemonico sta per Data Double (word)

segment .rodata
; è buona abitudine terminare le stringhe con il carattere ASCII zero
msg: db 'Ciao!',10,0 ; 10 è il carattere 'a capo'
msg_size equ $ - msg ; pseudo istruzione (macro); $ è l'indirizzo corrente

```

4 Somma con scanf e printf

Le chiamate di sistema sono impacchettate in librerie molto più comode da usare. Sfruttiamo perciò le librerie standard del C: `printf`, `exit` e `scanf`. È sufficiente dichiararne l'esistenza, ci penserà il **linker** a collegare effettivamente le istruzioni delle librerie.

```
extern scanf, printf, exit
```

Chiederemo all'utente del programma due numeri da sommare, conservandoli in due variabili x e y che è inutile inizializzare esplicitamente e che quindi possiamo mettere nel segmento `.bss`. Si tratta di un segmento speciale: lo spazio non viene preallocato, ma allocato solo a tempo di esecuzione (inizializzandolo a zero).

```
segment .bss
x: resd 1 ; reserve 1 double word
y: resd 1 ; reserve 1 double word
```

Servono anche le stringhe di formato per `printf` e `scanf`, come al solito in `.rodata`.

```
segment .rodata
msg: db 'Inserisci due numeri interi: ',0
ifmt: db '%d',0
ofmt: db 'Somma: %d',10,0
```

Per passare i parametri a una funzione basta metterli sullo **stack**, in ordine inverso.

```
push msg
call printf
```

Nelle `scanf` serve anche l'indirizzo della variabile in cui conservare i byte ricevuti dall'utente.

```
push x
push ifmt
call scanf
```

```
push y
push ifmt
call scanf
```

4.1 Il programma finale

```
extern scanf, printf, exit
global main
segment .text
main:

push msg
```

```

call printf

push x
push ifmt
call scanf

push y
push ifmt
call scanf

mov eax, [x]
add eax, [y]
push eax

push ofmt
call printf

push 0
call exit

segment .rodata
imsg: db 'Inserisci due numeri interi: ',0
ifmt: db '%d',0
ofmt: db 'Somma: %d',10,0
segment .bss
x: resd 1 ; reserve 1 double word
y: resd 1 ; reserve 1 double word

```