Sistemi Operativi

Laboratorio – linea 2

2

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
- Setup lab
- Qemu
- Astrazioni

Chiamate implicite
- Editor

Matteo Re

Dip. di Informatica
Università degli studi di Milano

matteo.re@di.it

a.a. 2017/2018 – Sem. II

http://homes.di.unimi.it/re/solabL2.html

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Lezione 2: Linguaggio Assembly & System calls

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Nella lezione precedente abbiamo cercato di rispondere alla domanda :

“Perchè serve un sistema operativo?”

Siamo giunti alla conclusione che un s.o. Rende molto più conveniente l'utilizzo della macchina fisica (sia da parte dell'utente finale che da parte del programmatore).

Per riuscirci il s.o. :

- fornisce un insieme di astrazioni che **semplificano l'uso di periferiche e memoria**.
- Ripartisce opportunamente le risorse fra tutte le attività in corso.

Sistemi Operativi
Bruschi
Monga
Re
Astrazioni del s.o.
Ruolo del s.o.
Setup lab
Qemu
Astrazioni
Chiamate implicite
Editor

Le principali astrazioni fornite dal sistema operativo sono:

- System call
- Memoria virtuale
- Processo
- File
- Shell

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

SYSCALL

Una chiamata di sistema (**syscall**) è la richiesta di un servizio al sistema operativo, che la porterà a termine in conformità alle sue *politiche*.

Per il programmatore è analoga ad una chiamata di procedura. Eneralmente viene realizzata mediante un'*interruzione software* per garantire la protezione del s.o.

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Un'interruzione (***interrupt request*** (IRQ)) è un segnale (solitamente generato da una periferica, ma non solo) che viene notificato alla CPU. La CPU, secondo le politiche programmate nel PIC, risponderà all'interruzione eseguendo il codice del *gestore dell'interruzione* (interrupt handler).
Dal punto di vista del programmatore la generazione di un'IRQ è analoga ad una chiamata di procedura ma:

- Il codice è totalmente disaccoppiato. Potenzialmente in uno spazio di indirizzamento diverso (permette le protezioni)
- Non occorre conoscere l'indirizzo della procedura
- La tempistica di esecuzione è affidata alla CPU

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

FILE

Un file è un insieme di byte conservati nella memoria di massa. Ad esso sono associati un **nome** ed altri attributi. Nei sistemi unix-like I file sono organizzati gerarchicamente in directory (l'equivalente dei folder in MS Windows), che non sono che altri file contenenti un elenco...

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Editor di testo

Un editor è un programma che permette di modificare arbitrariamente un *file*. Un editor di testo, generalmente, manipola file composti da caratteri stampabili.

Ne esistono moltissimi tipi:

UNIX-like o.s. : **vi**, emacs, nano, …
MS Windows : Notepad, Textpad, ...

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Editor vi

Scritto da Bill Joy (cofondatore Sun), 1976, per BSD UNIX.
E' un editor **modale**, nel senso che opera in diverse **modalità**, ed è quindi importante saper passare da una all'altra.

Le due modalità principali sono:
- modo **input**
- modo **comandi**

- I comandi di movimento e modifica sono sostanzialmente ortogonali
- E' un editor piccolo e veloce
- Fa parte dello standard POSIX

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Salvare un file e uscire wq

vi in una slide

- Modifica:
  - i,a insert before/after
  - o,O add a line
  - d,c,r delete, change, replace
  - y,p "to yank" and paste
  - u undo . redo
  - s/reg/rep/[g] search and replace
- Movimento:
  - h,j,k,l (o frecce)
  - 0, beginning of line, $, end of line
  - w, beginning of word, e, end of word
  - (num)G, goto line num, /, search
  - (,), sentence

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

SHELL

La shell è un <u>interprete dei comandi</u> che l'utente dà al sistema operativo. Ne esistono di grafiche e di testuali.

In ambiente GNU/Linux la più diffusa è una <u>shell testuale</u> **bash**, che fornisce I costrutti di base di un linguaggio di programmazione (variabili, strutture di controllo) e primitive per la gestione dei processi e dei file.

OBIETTIVI DI OGGI:

Sappiamo rispondere alla domanda “Perchè è necessario un sistema operativo”. Sappiamo che il s.o. fornisce una serie di astrazioni che rendono più conveniente l'uso dello hardware.

Oggi ci concentriamo sulla parola “**fornisce**”...
Cosa intendiamo quando diciamo che il sistema operativo **fornisce** astrazioni? E soprattutto … come si fa ad usufruire di queste astrazioni?

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

System calls

(e altro)

in Assembly

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Assembly è un linguaggio di programmazione minimale. Se vogliamo scrivere programmi utili in assembly molto spesso la via più semplice è quella di sfruttare I servizi forniti dal sistema operativo ed efettuare delle **chiamate di sistema** (system calls).

Esse sono librerie di funzioni incluse nel s.o. che semplificano task quali leggere un input dalla tastiera o scrivere qualcosa sullo schermo.

La lezione di oggi è dedicata alle chiamate di sistema.

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Quando utilizziamo una chiamata di sistema il sistema operativo sospende immediatamente l'esecuzione del nostro programma, quindi contatta i driver necessari ad eseguire l'operazione richiesta.

Una volta completato il task il s.o. restituisce il controllo al programma che ha effettuato la chiamata di sistema.

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Schema generale chiamata di sistema:

1) Caricare in **EAX** il numero identificativo di funzione per la quale richiediamo l'esecuzione da parte del s.o.

2) Caricare in altri registri gli argomenti della funzione. Attenzione: il numero di argomenti ed I registri in cui vanno caricati **cambia** da funzione a funzione!

3) Ultilizare l'istruzione **INT** per inviare una **interruzione software**. A questo punto il s.o. prende il controllo ed esegue la funzione utilizzando gli argomenti caricati nei registri.

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Scelta della funzione da eseguire:

Se richiediamo una interruzione quando EAX=**1** verrà eseguita **sys_exit**

Se richiediamo una interruzione quando EAX=**4** verrà eseguita **sys_write**

I registri utilizzati per passare gli (eventuali) argomenti sono **EBX, ECX** e **EDX**.

Esempio di System call table (Linux) :
http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

STRUTTURA SORGENTI ASSEMBLY:

Nei sorgenti assembly troviamo spesso la definizione di diverse sezioni (.data, .text, .global e .bss (per dichiarazione variabili)):

.data

Contiene dati inizializzati o costanti.

.text

Contiene il codice assembly. E' read-only (miglior protezione) il s.o. ne carica un'unica copia in memoria (risparmio memoria). Istanze diverse del programma condividono il contenuto di questa sezione.

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

OTTENERE I SORGENTI DEGLI ESERCIZI:

(all'interno di QEMU)

Scrivete ed eseguite I seguenti comandi in QEMU:

```
export http_proxy=proxy-aule-1.unimi.it:8080

wget http://homes.di.unimi.it/re/Corsi/SOLAB2_1314/asm_examples.tar.gz
```

Una volta ottenuto il file asm_examples.tar.gz estraetene il contenuto:

```
tar -xvzf asm_examples.tar.gz
```

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

OTTENERE I SORGENTI DEGLI ESERCIZI:

(all'interno di QEMU)

ATTENZIONE … PERICOLO …

Non ho fornito I sorgenti perchè vengano utilizzati così come sono senza essere obbligati a scrivere nemmeno una riga di codice.
Li ho inclusi nel materiale della lezione come riferimento nel caso di un'assenza imprevista o nel caso in cui non siate riusciti a completare Il sorgente prima del momento in cui mostro la soluzione dell'esercizio.

PER SUPERARE L'ESAME (che viene svolto in QEMU) dovete essere in grado di muovervi **velocemente** con riga di comando e editor (in particolare per la parte inerente al JOS kernel). Quindi …

SFORZATEVI DI RIUSCIRE A COMPLETARE I SORGENTI PROIETTATI A LEZIONE IN TEMPO PER VERIFICARNE IL FUNZIONAMENTO **INSIEME** IN CLASSE

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

L2 E1
(lez. 2 esercizio 1)

Obiettivo: scrivere ciao mondo sullo schermo

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Creiamo una variable '**msg**' nella sezione .data ed assegniamo ad essa una stringa, in questo caso '**Hello, world!**'.
Nella sezione .text diremo al s.o. dove iniziare l'esecuzione inserendo una etichetta globale **_start** : essa rappresenta il punto di ingresso nell'esecuzione del programma.

Useremo **sys_write** per stampare il messaggio sullo schermo. Il codice funzione da usare è **4** in Linux. La funzione richiede 3 argomenti da caricare in **EDX**, **ECX** e **EBX** prima di richiedere l'interruzione.

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Argomenti **sys_write**:

In EDX : lunghezza (in byte) della stringa.

In ECX : indirizzo della variabile creata in **.data**

In EBX : Indicatore file su cui si vuole scrivere – in questo caso **STDOUT** (che è associato allo schermo)

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

helloworld.asm

NB: solo se siamo in un sistema 64 bit

```
; Hello World
; Compilazione: nasm -f elf helloworld.asm
; Link: ld -m elf_i386 helloworld.o -o helloworld
; Run: ./helloworld

SECTION .data
msg     db      'Hello World!', 0Ah     ; assign msg variable with your message
string

SECTION .text
global  _start

_start:

    mov     edx, 13     ; number of bytes to write – string len and 0Ah
    mov     ecx, msg   ; move the memory address of our message string into ecx
    mov     ebx, 1      ; write to the STDOUT file
    mov     eax, 4      ; invoke SYS_WRITE (kernel opcode 4)
    int     80h
```

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

L2 E2
(lez. 2 esercizio 2)

Obiettivo: scrivere 'Hello World!' sullo schermo … senza provocare un errore di segmentazione

Sistemi Operativi

Laboratorio – linea 2

2

Abbiamo visto l'utilizzo di sys_write. Ora crercheremo di utilizzare una delle più importanti tra le chiamate di sistema: **sys_exit**.

Come mai, dopo aver eseguito il programma dell'esercizio precedente ottenevamo un errore di segmentazione?

I programmi eseguibili su un calcolatore possono essere immaginati come una lunga serie di istruzioni caricate in memoria e divise in sezioni (segmenti). La 'memoria', intesa genericamente, è condivisa da tutti I programmi e può essere utilizzata per lo storage di variabili, istruzioni … altri programmi.

Ogni porzione di memoria ha il suo indirizzo in modo che le informazioni possano essere ritrovate nel momento in cui si rendono necessarie.

Sistemi Operativi

Laboratorio – linea 2

2

Per eseguire un programma caricato in memoria utilizziamo l'etichetta globale _start: diciamo al s.o. La posizione, in memoria, in cui il programma può essere trovato ed eseguito.

Viene quindi effettuato un accesso alla memoria e questa viene letta in modo sequenziale seguendo la logica del programma (che determina l'indirizzo della prossima istruzione da eseguire).

Il s.o. salta all'indirizzo di "inizio esecuzione" del programma e procede da lì.

PROBLEMA: E' importante dire al s.o. non solo dove inizia il programma … ma anche dove **finisce**! Nell'esempio precedente **non** l'abbiamo fatto. Quindi, arrivato alla fine della regione di memoria in cui il programma è contenuto il s.o. è andato avanti e ha "sconfinato" in qualcos'altro (non sappiamo cosa). Il s.o. se ne è accorto e ha sollevato un a eccezione.

Sistemi Operativi

Laboratorio – linea 2

2

La chiamata di sistema che serve a specificare la fine (conclusione) di un programma è **sys_exit**.

La sua deefinizione è semplice. In Linux il suo codice numerico è **1** e ha un solo argomento da caricare in EBX.

Per richiedere l'esecuzione di questa funzione dobbiamo:

In EBX : caricare il valore **0** ('zero errors')
In EAX : caricare il codice numerico di sys_exit, **1**
Richiedere una interruzione (INT 80h).

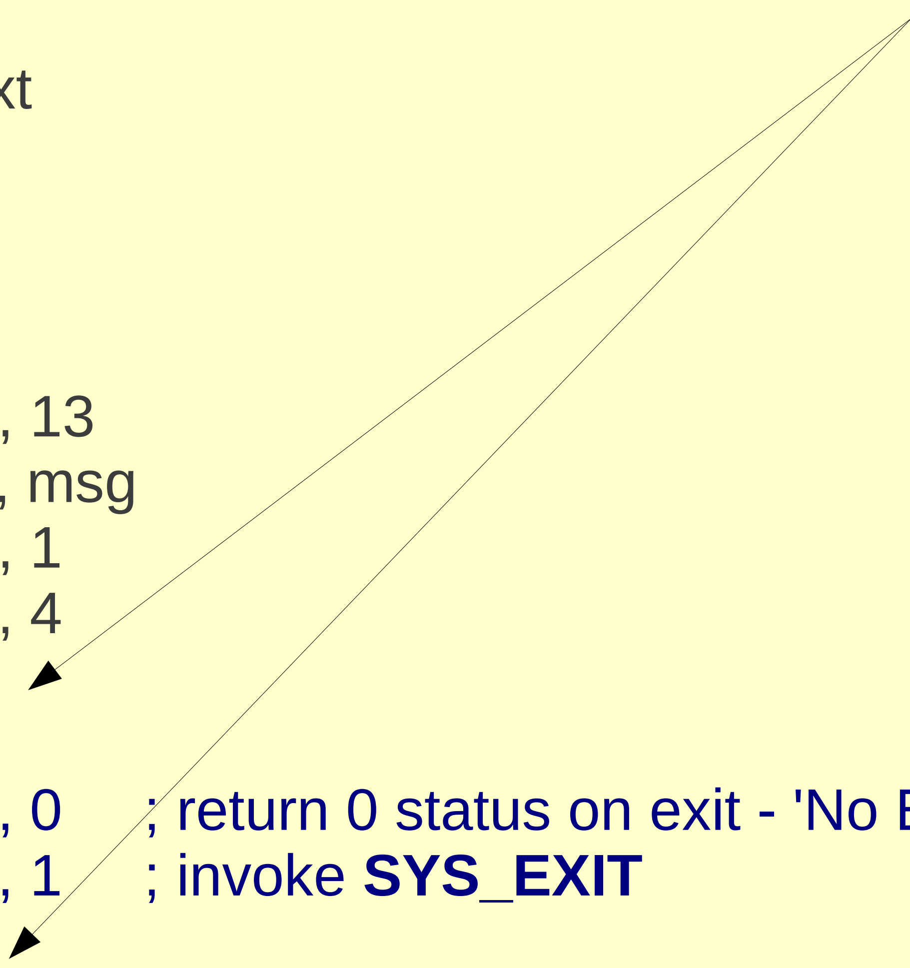
Sistemi Operativi

Laboratorio – linea 2

2

helloworld.asm

```
SECTION .data
msg     db      'Hello World!', 0Ah

SECTION .text
global  _start

_start:

    mov     edx, 13
    mov     ecx, msg
    mov     ebx, 1
    mov     eax, 4
    int     80h

    mov     ebx, 0      ; return 0 status on exit - 'No Errors'
    mov     eax, 1      ; invoke SYS_EXIT
    int     80h
```



NB: una interruzione per ogni syscall

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

L2 E3
(lez. 2 esercizio 3)

Obiettivo: scrivere un programma in cui sia coinvolto il calcolo della lunghezza di una stringa. Utilizzo di label.

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Perchè dovrebbe essere utile calcolare la lunghezza di una stringa (rispetto all'uso delle chiamate di sistema)?

sys_write richiede un puntatore all'indirizzo di memoria in cui risiede la stringa da scrivere e la lunghezza (in byte) del messaggio che vogliamo scrivere. Se modifichiamo la stringa dovremo aggiornare il numero dei byte da scrivere passato come argomento a sys_write, altrimenti questa non potrà comportarsi in modo corretto.

Esperimento: modificare il sorgente L2 E2 cambiando il contenuto della stringa in msg da 'Hello world!' a 'Hello solab2 world!''. Cosa stampa?

Avere un modo per calcolare la lunghezza di una stringa non è solo utile in generale … in alcuni casi è indispensabile (ad esempio se vogliamo stampare un input fornito dall'utente.

Sistemi Operativi

Laboratorio – linea 2

2

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Per calcolare la lunghezza di una generica stringa utilizzeremo una tecnica basata sull'**aritmetica dei puntatori**.

Il piano è questo:

1) **2** registri vengono inizializzati per puntare all'indirizzo in memoria in cui inizia la stessa stringa
2) Il contenuto di uno solo dei due registri (in questo caso **EAX**) sarà incrementato di un byte **fino al raggiungimento della fine della stringa**.
3) Sottrarre dal valore in EAX (indirizzo fine stringa) il valore contenuto nel secondo registro (indirizzo inizio stringa)
4) Questo equivale, effettivamente, a effettuare una sottrazione tra due array. Il risultato è il numero di elementi compresi tra I due indirizzi.
5) Il valore calcolato è passato come argomento a sys_write

Capire quando la stringa finisce

NB:
c.f.r. Solab2 slide LEZ1 per maggiori info su **EFLAGS** (che contiene ZF)

L'istruzione Assembly **CMP** confronta i suoi argomenti e imposta alcuni flag che possono essere utilizzati in operazioni di controllo di flusso unitamente ad intruzioni che realizzano dei salti (jump) condizionali.

Il flag di cui testeremo il contenuto si chiama **ZF** o Zero Flag. Quando il bite presente in memoria all'indirizzo contenuto in EAX contiene 0 (carattere di fine stringa) ZF viene impostato da CMP a 1.Useremo quindi l'istruzione **JZ** (**Jump if Zero**) per saltare (**quando** ZF contiene 1) ad un punto nel nostro programma identificato da un'etichetta: '**finished**'. Questo ci permetterà di uscire dal ciclo che incrementa il valore di EAX fino a portarlo all'indirizzo in cui finisce la stringa in msg (ciclo **nextchar**).

helloworld-len.asm 1/2

```
SECTION .data
msg     db      'Hello, solab2 world!', 0Ah ;

SECTION .text
global  _start

_start:
   mov     ebx, msg      ; move the address of our message string into EBX
   mov     eax, ebx      ; move the address in EBX into EAX

nextchar:
    cmp     byte [eax], 0   ; compare the byte pointed to by EAX at this address against zero
    jz      finished        ; jump if ZF to the point in the code labeled 'finished'
    inc     eax             ; increment the address in EAX by one byte
    jmp     nextchar        ; jump to the point in the code labeled 'nextchar'

finished:
    sub     eax, ebx   ; subtract the address in EBX from the address in EAX (result stored in EAX)
```

helloworld-len.asm 2/2

```
mov     edx, eax        ; EAX now equals the number of bytes in our string
mov     ecx, msg        ; standard sys_write call
mov     ebx, 1
mov     eax, 4
int     80h

mov     ebx, 0          ; sys_exit call
mov     eax, 1
int     80h
```

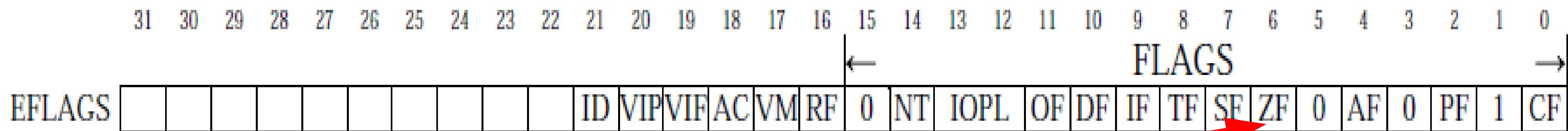
Sistemi Operativi

Laboratorio – linea 2

2

MACCHINA i386 : register set

Flags



Bit		Flag	Description
0	CF	Carry Flag	Carry from most significant bit, also borrow for most significant bit; can be considered as overflow in unsigned instructions.
2	PF	Parity Flag	Set to 1 if 8 less significant bits of result have even number of 1's, else set to 0.
4	AF	Auxiliary carry Flag	Used as carry flag in BCD instructions.
6	ZF	Zero Flag	Set to 1 if result is zero, else set to 0.
7	SF	Sign Flag	Set to 1 if result is negative (below zero), else set to 0.
8	TF	Trap Flag	Used by debuggers.
9	IF	Interrupt Flag	If set to 1, then interrupts are enabled, else are disabled.
10	DF	Direction Flag	When set to 0, string instructions increment the index registers, else – decrement the index registers.
11	OF	Overflow Flag	Used in signed instructions.
12,13	IOPL	I/O Privilege Level	Indicates the I/O privilege level of the currently running program or task.
14	NT	Nested Task	Controls the chaining of interrupt and called tasks.
16	RF	Resume Flag	Controls the processor's response to instruction-breakpoint conditions.
17	VM	Virtual 8086 Mode	Set to enable virtual-8086 mode; clear to return to protected mode.
18	AC	Alignment check	Set this flag and the AM flag in control register CR0 to enable alignment checking of memory references.
19	VIF	Virtual Interrupt Flag	Contains a virtual image of the IF flag.
20	VIP	Virtual Interrupt Pending	Set by software to indicate that an interrupt is pending.
21	ID	Identification	The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction.

Sistemi Operativi
Bruschi
Monga
Re
Astrazioni del s.o.
Ruolo del s.o.
Setup lab
Qemu
Astrazioni
Chiamate implicite
Editor

L2 E4
(lez. 2 esercizio 4)

Obiettivo: riutilizzo del codice … subroutines

Una buona pratica di programmazione (riguardante anche altri linguaggi oltre ad Assembly) è il riutilizzo del codice.

In questo esempio vedremo come utilizzare le subroutine in Assembly. Esse vengono indicate tramite etichette ma **NON SI ESEGUONO MEDIANTE istruzioni della famiglia JMP**.

In questo esempio vedremo come dichiarare le subroutine e come "chiamarle". Se seguiremo alcune convenzioni Assembly

1) Prima di eseguire una subroutine memorizzerà l'indirizzo a cui ci troviamo al momento della chiamata
2) Eseguirà la subroutine
3) Ritornerà al punto in cui era stata chiamata la subrutine … e da lì continuerà l'esecuzione

Per riuscirci utilizzerà lo **STACK**

A proposito dello **STACK**

Lo stack è un tipo speciale di memoria. Dal punto di vista fisico è lo stesso tipo di memoria di quella che abbiamo usato fin qui … ma quello che cambia è il modo in cui viene utilizzata.

Lo stack è una struttura LIFO (last in first out). Funziona come una pila di piatti … l'ultimo piatto ad essere aggiunto sarà il primo ad essere prelevato.

In Assembly lo stack può contenere variabili, indirizzi di altri programmi e, in generale, dati per I quali è necessario uno storage temporaneo. In particolare, quando usiamo le subroutine, lo stack può essere usato per immagazzinare temporaneamente valori da ripristinare (nella loro posizione originale, ad es. un dato registro) quando l'esecuzione della subroutine sarà completata.

Aggiungere / prelevare valori dallo stack : PUSH / POP

Il valore di ogni registro che sarà utilizzato dalla subroutine che vogliamo chiamare dovrebbe essere messo nello stack utilizzando l'istruzione **PUSH**.

In questo modo, una volta che l'esecuzione della subroutine è terminana questi valori potranno essere ripristinati mediante l'utilizzo dell'istruzione **POP**.

L'effetto finale è che, se usiamo opportunamente PUSH/POP il valore presente nel registro sarà **LO STESSO** prima e dopo la chiamata della subrutine … indipendentemente da quello che fa la subroutine stessa. In definitiva possiamo evitare di preoccuparci di quello che fa la subroutine con I registri … perchè questi saranno ripristinati dopo la sua esecuzione!

Chiamata di subroutine (e ritorno da essa)

Abbiamo detto che **le subroutine sono definite tramite etichette**. Verrebbe quindi la tentazione di saltare ad esse utilizzando una funzione di salto (magari un salto incondizionato, istruzione JMP). Ma, di solito, questo non si fa.

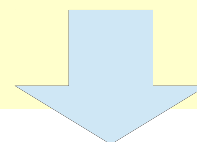
Si utilizzano, invece, le funzioni **CALL** e **RET** per, rispettivamente, chiamare e ritornare da una subroutine. Come mai?

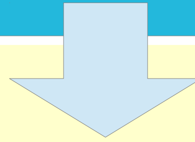
CALL e RET utilizzano, in modo del tutto **automatico**, lo stack. Quando usiamo **CALL** l'indirizzo del punto in cui viene effettuata la chiamata viene messo sullo stack. Quando l'esecuzione della routine è terminata se usamo **RET** l'indirizzo sullo stack viene prelevato e il programma ritorna al punto immediatamente succesivo alla chiamata. Questo è il motivo per cui non si dovrebbe mai “saltare” ad una subroutine ma utilizzare CALL/RET.

helloworld-len-sub.asm 1/2

```
SECTION .data
msg     db      'Hello, solab2 world!', 0Ah

SECTION .text
global  _start

_start:
    mov     eax, msg        ; move the address of our message string into EAX
    call    strlen          ; call our function to calculate the length of the string

    mov     edx, eax        ; our function leaves the result in EAX
    mov     ecx, msg        ; this is all the same as before
    mov     ebx, 1
    mov     eax, 4
    int     80h

    mov     ebx, 0
    mov     eax, 1
    int     80h
```

Aggiungeremo di seguito il codice della subroutine





helloworld-len-sub.asm 2/2

```
strlen:             ; this is our first function declaration
    push   ebx      ; push the value in EBX onto the stack to preserve it while we use EBX in this function
    mov    ebx, eax ; move the address in EAX into EBX (Both point to the same segment in memory)

nextchar:           ; this is the same as L2 E3 (this is a normal label)
    cmp    byte [eax], 0
    jz     finished
    inc    eax
    jmp    nextchar

finished:           ; this is the same as L2 E3 (this is a normal label)
    sub    eax, ebx
    pop    ebx      ; pop the value on the stack back into EBX
    ret             ; return to where the function was called
```

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

L2 E5
(lez. 2 esercizio 5)

Obiettivo: Il codice degli esempi sta diventando troppo lungo per essere inserito in un'unica slide … mettiamo tutte le funzioni in un file esterno e vediamo come includere questo file in un sorgente Assembly.

NB:Le prossime slide sono dedicate, ognuna, ad una funzione. Scrivetele tutte in un file di nome **functions.asm**

da scrivere in functions.asm

```
; ---------------------------------------
; int slen(String message)
; String length calculation function
slen:
    push    ebx
    mov     ebx, eax

nextchar:
    cmp     byte [eax], 0
    jz      finished
    inc     eax
    jmp     nextchar

finished:
    sub     eax, ebx
    pop     ebx
    ret
```

da scrivere in functions.asm

```
; void sprint(String message);String printing function
sprint:
    push    edx
    push    ecx
    push    ebx
    push    eax
    call    slen

    mov     edx, eax
    pop     eax

    mov     ecx, eax
    mov     ebx, 1
    mov     eax, 4
    int     80h

    pop     ebx
    pop     ecx
    pop     edx
    ret
```

da scrivere in functions.asm

```
;------------------------------------------
; void exit()
; Exit program and restore resources
quit:
    mov     ebx, 0
    mov     eax, 1
    int     80h
    ret
```

Sistemi Operativi
Bruschi
Monga
Re
Astrazioni del s.o.
Ruolo del s.o.
Setup lab
Qemu
Astrazioni
Chiamate implicite
Editor

Ora vediamo come includere il file functions.asm in un secondo file di nome helloworld-inc.asm

NB:
Nei prossimi esempi **NON mostrerò più il codice contenuto in functions.asm** (esso verrà semplicemente incluso) se non nel caso in cui sia necessario apportare delle modifiche ad esso (in particolare ad alcune funzioni). In questo caso segnalerò I cambiamenti necessari e mostrerò il codice come dovrà essere **dopo** la modifica.

helloworld-inc.asm

```
%include      'functions.asm'    ; include our external file

SECTION .data
msg1   db    'Hello, solab2 world!', 0Ah              ; our first message string
msg2   db    'This is how we recycle in NASM.', 0Ah   ; our second message string

SECTION .text
global  _start

_start:
    mov   eax, msg1        ; move the address of our first message string into EAX
    call  sprint           ; call our string printing function

    mov   eax, msg2        ; move the address of our second message string into EAX
    call  sprint           ; call our string printing function

    call  quit             ; call our quit function
```

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Assembliamo, linkiamo e proviamo ad eseguire.

… Notate qualcosa di strano?
Cosa scrive il programma sullo schermo?
E' quello che vi aspettereste? (motivate la risposta)

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

L2 E6
(lez. 2 esercizio 6)

Obiettivo: Capire cosa è andato storto in esercizio 5. E' più semplice di quello che sembra ma ci fornisce un'ulteriore occasione per capire come funziona Assembly...

byte di terminazione nulli

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Come mai, in L2 E5, la seconda stringa è stata stampata 2 volte? In realtà sprint (la funzione di stampa) è stata chiamata solo 2 volte e non 3...

L'output di E5 dovrebbe essere il seguente:

Hello, solab2 world!
This is how we recycle in NASM.

Esperimento:
Provate a commentare la seconda chiamata a sprint. Assemblate, linkate e provate ad eseguire. Come cambia l'output? Quali sono le vostre conclusioni a riguardo?

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Questo comportamento è dovuto al fatto che in L2 E5 non abbiamo terminato in modo appropriato le stringhe.

In Assembly le variabili sono immagazzinate in memoria una dopo l'altra in modo sequenziale e quindi l'ultimo byte di una variabile (ad es. msg1) è immediatamente prima del primo di quella successiva (ad es. msg2).

Sappiamo che la nostra funzione che calcola la lunghezza delle stringhe **cerca la prima occorrenza di un byte a valore 0 per capire quando la stringa è finita**. E se non lo trova cosa fa? **Va avanti** … e considera msg2 come parte di msg1.
Soluzione: terminare ogni stringa con un byte a valore 0. Modifichiamo il sorgente di L2 E5 come segue ...

helloworld-inc.asm

```
%include      'functions.asm'     ; include our external file

SECTION .data
msg1    db      'Hello, solab2 world!', 0Ah, 0h                 ; our first message string
msg2    db      'This is how we recycle in NASM.', 0Ah, 0h      ; our second message string

SECTION .text
global  _start

_start:
    mov   eax, msg1         ; move the address of our first message string into EAX
    call  sprint            ; call our string printing function

    mov   eax, msg2         ; move the address of our second message string into EAX
    call  sprint            ; call our string printing function

    call  quit              ; call our quit function
```

byte di terminazione nulli

L2 E7
(lez. 2 esercizio 7)

Obiettivo: Includere una funzione di libreria. Cercheremo di utilizzare una funzione esterna proveniente dalla libreria standard del C. La funzione che vogliamo usare è **printf** (come è possibile intuire serve a stampare qualcosa).
Inoltre cercheremo di capire se ci sono differenze rispetto ad un programma che utilizza solo codice Assembly scritto da noi o chiamate di sistema. Utilizzeremo il debugger GDB.

Sistemi Operativi

Laboratorio – linea 2

2

syscallext.asm

```
segment .text
        global main
        extern printf

main:
        push msg
        call printf            ; chiamata di libreria

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

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

segment .rodata
msg db 'Ciao solabbisti!',10,0
msg_size equ $ - msg
```

Stampa mediante funzione printf

stampa mediante **sys_write**

Calcolo automatico lunghezza stringa in msg

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Per assemblare:

nasm -f elf **-gstabs** syscallext.asm

Questa è una i maiuscola

Questa è una elle minuscola

Link (2 opzioni) :

In sistemi 32 bit (e in solab-live)
ld -I /lib/ld-linux.so.2 -lc -e main -o syscallext syscallext.o

In sistemi 64 bit (in cui abbiamo assemblato usando -I elf)
ld -m elf_i386 -I /lib/i386-linux-gnu/ld-linux.so.2 -lc -e main -o syscallext syscallext.o

NB: richiede installazione di package gcc-multilib

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

Chiamate implicite
Editor

Eseguire un programma nel debugger GDB:

```
$ gdb syscallext
```

```
(gdb) break main
(gdb) run
```

Esaminiamo il contenuto della memoria in corrispondenza del simbolo main (mostrando le istruzioni) :

(gdb) x/10**i** main

Cosa notate? Che istruzione c'è prima del nome della funzione printf (posto di lato ad un indirizzo di memoria) ?

Il simbolo printf è definito. Provate a scrivere
(gdb) disas printf
Osservazioni?

Sistemi Operativi

Bruschi
Monga
Re

Astrazioni del s.o.

Ruolo del s.o.
Setup lab
Qemu
Astrazioni

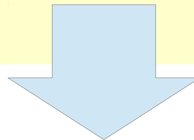
Chiamate implicite
Editor

L2 E8
(lez. 2 esercizio 8)

Obiettivo: Chiedere all'utente di inserire un valore (in questo esempio una stringa di testo) ed utilizzare il valore per fare qualcosa (in questo caso stampare un messaggio sullo schermo)

helloworld-input.asm 1/2

```
%include      'functions.asm'

SECTION .data
msg1     db     'Please enter your name: ', 0h     ; message string asking user for input
msg2     db     'Hello, ', 0h                      ; message string to use after user has entered his/her name

SECTION .bss
sinput:    resb    255                              ; reserve a 255 byte space in memory for the users input string

SECTION .text
global  _start

_start:
```



continua nella slide successiva ...

helloworld-input.asm 1/2

```
mov     eax, msg1
call    sprint

mov     edx, 255        ; number of bytes to read
mov     ecx, sinput     ; reserved space to store our input (buffer)
mov     ebx, 0          ; write to the STDIN file
mov     eax, 3          ; invoke SYS_READ (kernel opcode 3)
int     80h

mov     eax, msg2
call    sprint

mov     eax, sinput     ; move our buffer into eax (Note: input contains a linefeed)
call    sprint          ; call our print function

call    quit
```