

SISTEMI DI CALCOLO

9 CFU A.A. 2018/19

Camil Demetrescu

Emilio Coppa

Claudio Ciccotelli

migno.gioele@gmail.com

PRECEDENZA DEGLI OPERATORI IN C

La seguente tavola delle precedenze degli operatori in C dovrebbe essere sempre a portata di mano. Soprattutto quando si sta imparando il linguaggio di programmazione. Ma, anche i più esperti potranno trarre beneficio dall'avere riassunti in una unica tabella tutti gli operatori con le loro precedenze e associatività (= ordine di precedenza che viene seguito quando si trovano più operatori di uguale precedenza nella stessa espressione).

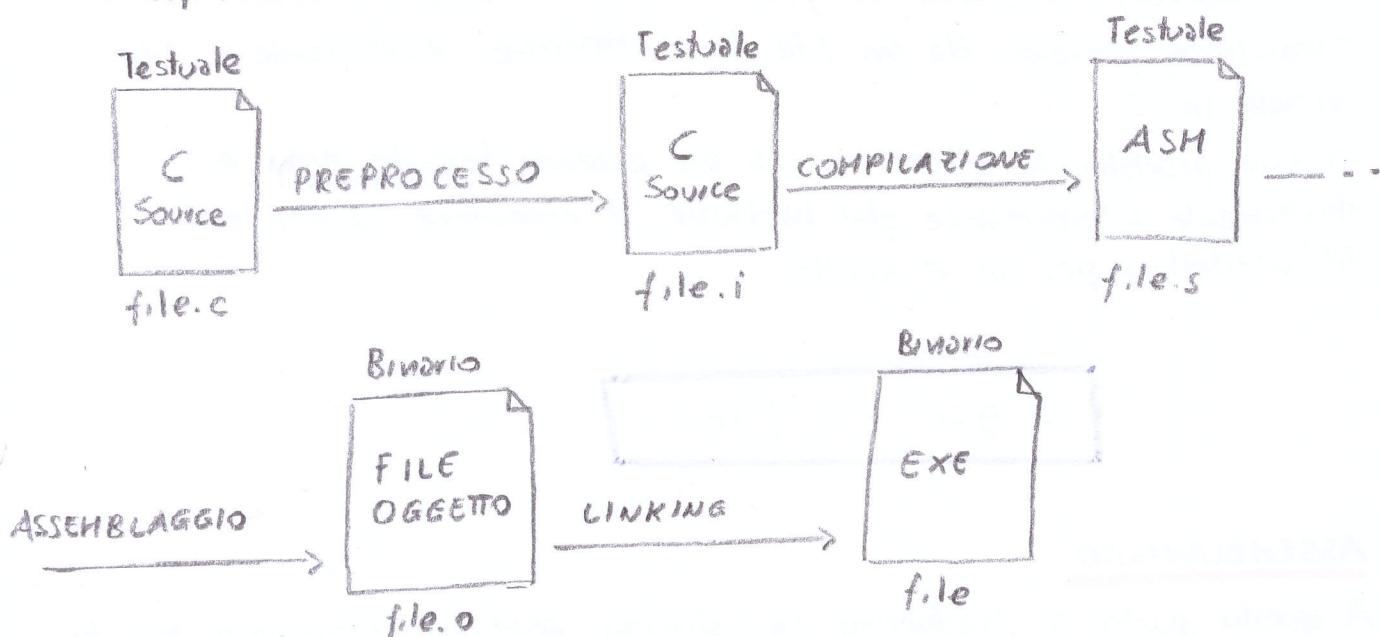
Operatore	Descrizione	Associatività
()	Parentesi (chiamata a funzione) (vedi Nota 1)	
[]	Parentesi quadre (array)	
.	selezione di membri di oggetti dal nome	
->	come sopra, ma dal puntatore	
++ --	incrementi/decrementi postfissi (vedi Nota 2)	
++ --	incrementi/decrementi prefissi	
+ -	più/meno unari	
! ~	negazione logia/complemento bit a bit	
(type)	cast (conversione di tipi)	
*	deferenza	
&	indirizzo (di operandi)	
sizeof	dimensioni in byte (dipendente dalla piattaforma)	
* / %	moltiplicazione/divisione/modulo	da sx a dx
+ -	Addizione/sottrazione	da sx a dx
<< >>	Shift a sinistra/destra bit a bit	da sx a dx
< <=	operatori relazionali minore di/minore o uguale a	
> >=	operatori relazionali maggiore di/maggiore o uguale a	
== !=	operatore relazionale uguale a/ non uguale a	da sx a dx
&	AND bit a bit	da sx a dx
^	OR esclusivo bit a bit	da sx a dx
	OR bit a bit	da sx a dx
&&	AND logico	da sx a dx
	OR logico	da sx a dx
? :	condizionale ternario	da dx a sx
=	assegnazione	
+= -=	assegnazione con addizione/sottrazione	
*= /=	assegnazione con moltiplicazione/divisione	
%= &=	assegnazione con modulo/AND bit a bit	
^= =	assegnazione con OR esclusivo bit a bit/Or bit a bit	
<<= >>=	assegnazione con shift a sinistra/shift a destra bit abit	
,	virgola (separatore di espressioni)	da sx a dx

Nota 1:Le parentesi sono usate anche per raggruppare sottoespressioni per forzare le precedenze. Nel caso di parentesi nidificate, l'ordine di precedenza è dalla più interna alla più esterna.

Note 2:Gli operatori di incremento postfissi hanno alta precedenza, ma l'incremento/decremto reale è rimandato fino alla fine della valutazione dell'espressione. Ad esempio nell'esecuzione di `a=3*i++`, se `i` valeva 1 prima di questa istruzione, `a` varrà 3 e `i` varrà 2. Cioè `i` è incrementato al termine dell'esecuzione dell'istruzione.

PROCESSO DI COMPILAZIONE FILE C (PIPELINE)

Al momento della compilazione di un file sorgente C si eseguono diverse fasi prima di ottenere l'eseguibile finale che è interpretabile direttamente dal hardware.



Si osservi che tutti questi step intermedii sono eseguiti in automatico dal compilatore gcc:

> gcc -o file file.c

Vediamo le diverse fasi includendo il comando gcc per eseguire il singolo step

• PREPROCESSO

In questa fase il compilatore esegue le direttive, ovvero i comandi preceduti da `#` producendo un file c con estensione .i
es:

```

# include <stdio.h>
#define COST 1
  
```

Nel caso di `#include`, il compilatore copia nel nuovo file l'intero contenuto del file importato.

> gcc -E file.c > file.i

OSS

[L'estensione del file di output (file.i) Non è uno standard]

• COMPILAZIONE

In questa fase il compilatore traduce le istruzioni C, scritte in file.i, in Assembly. Un'istruzione C si traduce in n istruzioni in Assembly, a causa di questa relazione NON BIUNIVOCÀ risulta IMPOSSIBILE, partendo da un file ASH, ottenere esattamente il file iniziale in C.

Il file prodotto ha estensione .s ed è essenzialmente non è direttamente interpretabile dal hardware, è comunque dipendente dall'architettura per cui è scritto.

```
> gcc -s file.i
```

• ASSEMBLAMENTO

A questo punto si traducono le istruzioni ASH in linguaggio macchina producendo un file binario detto FILE OGGETTO, con estensione .o.

In questo caso la corrispondenza tra i due linguaggi è BIUNIVOCÀ, quindi è facile passare da uno all'altro, il processo inverso del DISASSEMBLAGGIO è immediato.

Si osservi che il file ottenuto NON È UN FILE DIRETTAMENTE ESEGUIBILE dato che contiene solo i comandi inclusi nel file scritto, necessari per eseguire il programma, ma NON È SUFFICIENTE per far partire un processo, occorre interfacciarsi correttamente con il sistema operativo.

```
> gcc -c file.s
```

Per disassemblare il file oggetto generato dalla precedente istruzione possiamo usare il seguente comando:

```
> objdump -d file.o
```

• LINKING

In questa fase si procede all'unione di tutti i file oggetto necessari per eseguire il programma con l'aggiunta del codice detto GLUECODE, necessario per collegare i diversi file.

```
> gcc -o file file.o
```

INTRODUZIONE ASSEMBLY

I sistemi di calcolo si basano su diverse ASTRATTIONI che consentono di semplificare il funzionamento della macchina evitando alcuni dettagli implementativi.

Le astrazioni più importanti sono:

- MEMORIA: vista come un grosso array di byte

- INSTRUCTION SET ARCHITECTURE (ISA):

Definisce:

- a. Stato CPU

- b. Formato delle sue istruzioni

- c. Effetto che le istruzioni hanno sullo stato (semantica)

Per tradurre codice di alto livello, ad esempio C, in codice macchina il compilatore si basa sulla descrizione astratta dell'architettura della macchina data dalla sua ISA.

Le due ISA più diffuse a livello consumer sono:

- IA32 (x86): Descrive le architetture dei processori x86 32 bit

- X86-64: Descrive le architetture dei processori x86 64 bit

L'ISA X86-64 è ottenuto da un'estensione della x86, quindi quest'ultima è compatibile con la versione a 64 bit

Nel nostro corso ci occuperemo di studiare la IA32 dato che è più utile a scopo didattico

• TIPI DI SINTASSI ASSEMBLY

I due tipi di sintassi assembly più diffusi sono:

- Intel : Sintassi che troviamo nei manuali Intel

- AT&T : Sintassi compatibile con ambiente UNIX

Questa divisione è dovuta principalmente ad un fatto storico, al momento dello sviluppo di GCC i suoi creatori hanno realizzato un compilatore che fosse compatibile con il sistema proprietario UNIX il quale era il più diffuso e sviluppato dall'azienda AT&T, inoltre le diverse architetture Intel x86 sono state sviluppate successivamente.

Nel nostro corso faremo riferimento alla sintassi AT&T la quale risulta più utile dal punto di vista didattico dato che in parte è più semplice ed inoltre esplicita alcuni vincoli che consentono di comprendere meglio il funzionamento del programma.

OSS (Traduzione sintassi Intel)

Dato che useremo AT&T, non potremo usare direttamente le istruzioni presenti nel manuale Intel, ma andranno leggermente modificate.

ESEMPIO (Introduzione ASH) 26F

Vediamo un esempio di codice C tradotto in ASH e poi proviamo a compilare direttamente un file ASH.

In questo esempio ci è utile il seguente comando che ci consente di comprendere di che tipo è un file:

```
> file hello.c
```

Partiamo dai seguenti file C:

```
// main.c
#include <stdio.h>
int foo();
int main()
{
    int res = foo();
    printf("%d\n", res);
}
```

```
// foo.c
int foo()
{
    int x;
    x=10;
    return x;
}
```

Procediamo alla compilazione:

> gcc -o main main.c foo.c

Tramite il seguente comando ottengo informazioni sull'eseguibile.

> file main

Ottengono che tale file binario è in formato ELF, tale formato identifico in ambiente Linux i file oggetto ed eseguibili, in Mac OS X si usa invece il formato Mach-O.

Procedendo al disassemblaggio dell'eseguibile main ottengono il codice macchina relazionato al codice ASH, in questo file è contenuto tutti i file e include anche il gluecode.

Vediamo invece il disassemblaggio del file oggetto foo.o:

> gcc -c foo.c

> objdump -d foo.o

Nell'output si ha il seguente layout:

ID Funzione	000000 <foo>:	Nome	push %rbp
	0: 55		mov %rsp,%rbp
	1: 48 89 e5		movl \$0x0,-0x4(%rbp)
	4: c7 45 fc 0a 00 00 00		:
	:	Istruzione in esadecimale, si noti che hanno dimensione variabili. Quelle più comuni occupano meno byte.	:
		Numero byte Inizio istruzione	Comando ASH
			Argomenti del comando

- foo.c → foo.s

Tramite il comando

> gcc -S foo.c

ottengo il file ASH (foo.s), tale file contiene diverse istruzioni precedute da un punto che le identifica come direttive per i processi di compilazione successivi.

L'unica direttrice fondamentale è'

• global foo

| "global" sta per globale

Tale direttiva è necessaria per dichiarare che il file foo.s contiene la definizione della funzione foo, il compilatore quindi al momento della lettura del main troverà il corpo della funzione in foo.s.

- File ASH (foo.s)

Vediamo come scrivere direttamente il file foo.s e compilarlo insieme a main.c

```
// foo.s
.globl foo
foo:
    movl $10, %eax
    ret
    movl %eax, -4(%ebp)
    movl -4(%ebp), %eax
    movl %eax, (%esp)
```

Annotations:

- \$ individua una costante
- % individua un registro dello CPU
- eax è il registro
- ret → Comando di ritorno al chiamante
- movl Sorgente, Destinatario
- il comando è
- l è una specifica

Il programma foo.s è leggermente diverso da quello che crea il compilatore, per semplicità non abbiamo usato una variabile, ma abbiamo caricato il valore 10 nel registro eax, tale registro è quello dedicato per ritornare il risultato della funzione al chiamante.

Uniamo tale file con main.c nel seguente modo:

```
> gcc -c foo.s
> gcc -c main.c
> gcc -o main main.o foo.o
```

O direttamente:

```
> gcc -o main main.c foo.s
```

oss (ASH X86)

Il codice scritto da noi in ASH è scritto per IA32, il compilatore di default invece produce un ASH per X86-64.

Per produrre un ASH per X86 32 bit si usa:

```
> gcc -m32 -S foo.c
```

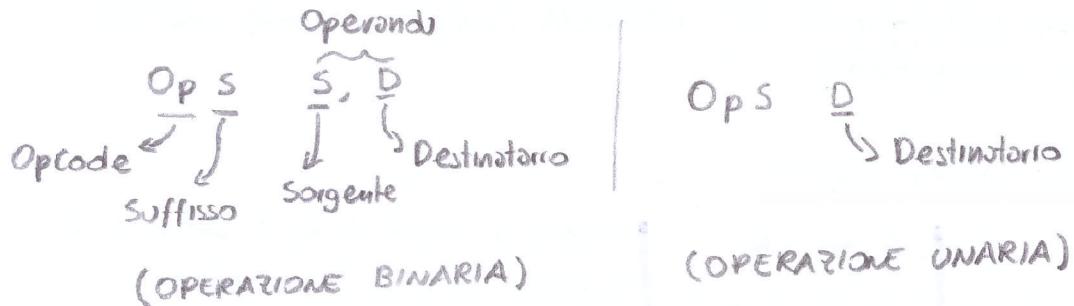
Oppure per ottenere direttamente un ELF 32 bit:

```
> gcc -m32 -o main main.c foo.c
```

In Kubuntu 18.04 LTS
non compila
#include
<bits/libc-header
-start.h>
NON ESISTENTE

STRUTTURA ISTRUZIONI AT&T

Le istruzioni ASH con sintassi AT&T possono essere BINARIE o UNARIE (due operandi oppure uno) ed hanno la seguente forma:



Analizziamo e classifichiamo gli elementi che compongono un istruzione ASH IA32 AT&T.

OPCODE

L'OPCODE identifica una FAMIGLIA di istruzioni e tramite l'aggiunta del suffisso se ne individua una specifica.

Le istruzioni si dividono in tre tipi:

SPOSTAMENTO DATI: In IAB32 sono consentiti gli spostamenti dati del tipo:

- Registro \leftrightarrow Registro
 - Registro \leftrightarrow Memoria

Non e' quindi consentito lo scambio diretto tra due celle di memoria, cio' puo' sembrare una limitazione, ma in altre architetture, ad esempio ARH, non e' consentito neanche tra registro e memoria. La motivazione per queste scelte e' dovuta alla relativa lentezza all'accesso in memoria che andrebbe a rendere eccessivamente lunga l'esecuzione di un'operazione.

- ARITMETICHE - LOGICHE

- SALTO (Controllo del flusso): Un esempio e' l'istruzione "ret" che fa ritornare il flusso del programma al chiamante della funzione.

Vediamo inizialmente alcune operazioni dei primi due tipi.

- Istruzioni spostamento dati

mov S,D | $D \leftarrow S$ | Copia byte dalla sorgente S alla destinazione D.

Si ricorda che come detto in precedenza i due operandi NON possono essere entrambi di tipo MEMORIA (successivamente vediamo i diversi tipi di operandi)

- Istruzioni Aritmetico-logiche

neg D | $D \leftarrow -D$

add S,D | $D \leftarrow D + S$

sub S,D | $D \leftarrow D - S$

imul S,D | $D \leftarrow D * S$

inc D | $D \leftarrow D + 1$

dec D | $D \leftarrow D - 1$

Inverte il segno di D, e salva in D

Somma D e S e memorizza il risultato in D

Sottrae a D il valore di S e memorizza il risultato in D

Moltiplica S e D, e salva il risultato in D. D DEVE ESSERE UN REGISTRO.

Incrementa D di uno e salva in D

Decrementa D di uno e salva in D

Si noti che l'operazione "imul" ha un vincolo sugli operandi, ovvero il destinatario DEVE ESSERE di tipo REGISTRO

or S,D | $D \leftarrow D \text{ OR } S$

OR Bitwise (Bit a Bit) tra D e S e salva in D

and S,D | $D \leftarrow D \& S$

AND Bitwise tra D e S e salva in D.

xor S,D | $D \leftarrow D \wedge S$

XOR Bitwise tra D e S e salva in D

not D | $D \leftarrow !D$

Inverte ogni bit di D

es: (Operazioni Bit a Bit)

D	1	0	1	0	1	1
S	1	1	1	1	0	1
D&S	1	0	1	0	0	1

Tabelle Verit`

A	B	A \wedge B	A & B	A $\wedge\wedge$ B	!A
0	0	0	0	0	1
0	1	0	0	0	1
1	0	0	0	0	0
1	1	1	1	1	0

OPERANDI

Le istruzioni hanno in genere uno o più operandi che definiscono i dati su cui operano. Solitamente se ne hanno due; l'**OPERANDO SORGENTE** che specifica un valore di ingresso per l'operazione ed un **OPERANDO DESTINATARIO** che identifica dove deve essere immagazzinato il risultato dell'operazione.

Gli operandi SORGENTE possono essere di tre tipi:

- **IMMEDIATO**: operando costante, ad esempio \$10.
- **REGISTRO**: operando memorizzato in un registro
- **MEMORIA**: operando memorizzato in memoria

Gli operandi DESTINATARIO ovviamente NON possono essere di tipo IMMEDIATO, sono quindi solo di due tipi:

- **REGISTRO**: il risultato dell'operazione viene memorizzato in un registro.
- **MEMORIA**: il risultato dell'operazione viene salvato in memoria

Si ricorda che oltre al vincolo del destinatario NON IMMEDIATO si ha il vincolo di un solo operando di tipo memoria per ogni operazione ed inoltre nel caso di "imul" il destinatario DEVE essere un registro.

Per descrivere la sintassi degli operandi, useremo le seguenti notazioni:

- Se "reg" è il nome di un registro $R[reg]$ denota il contenuto di reg.
- Se "x" è un indirizzo di memoria, $H_b[x]$ denota l'oggetto di b byte all'indirizzo x.

- Operandi Immediati

SINTASSI	VALORE DENOTATO	NAME CONVENZIONALE
\$ imm	imm	Immediato

es:

$$\$10 = 10$$

- Operandi Registri

SINTASSI	VALORE DENOTATO	NAME CONVENZIONALE
% reg	R[reg]	Registro

es:

$$\%eax$$

- Operandi Memoria

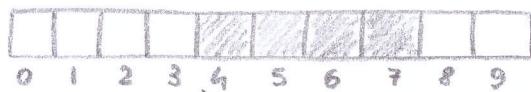
SINTASSI	VALORE DENOTATO	NAME CONVENZIONALE
imm	$M_b[imm]$	Assoluto
(reg)	$M_b[R[reg]]$	Indiretto
imm(reg)	$M_b[R[reg] + imm]$	Base e spostamento

Il numero di byte b della memoria viene specificato dal suffisso dell'istruzione, si ricorda che la memoria e' considerata come un array di byte.

es:

$$R[eax] = 2 \quad (\text{il contenuto di eax e' 2})$$

Memoria []



Con l'istruzione: $\text{movl } 2(\%eax), \%ecx$

Il suffisso "l" Indica 4 Byte

$$M[2+2 ; 2+2+4)$$

La Sorgente e'

Destinazione, si ricorda che deve essere un REGISTRO

SUFFISSI

L'IA32 ha diversi tipi di dato numerici primitivi (tipi di dato macchina), noi useremo i seguenti per cui corrisponde il suffisso Assembly per le operazioni:

Tipo di dato macchina	Rappresentazione	Suffisso	Byte	Tipo dato C
Byte	Intero	b	1	(unsigned) char
Word	Intero	w	2	(unsigned) short
Double Word	Intero	l	4	{(unsigned) int (unsigned) long}

OSS (unsigned - signed)

Interi con o senza segno hanno il medesimo tipo macchina corrispondente: ad esempio, sia char che unsigned char sono rappresentati come Byte.

OSS (Studio dati, Interi)

In questo corso ci concentreremo unicamente sui dati di tipo INTERO e non su quelli in virgola mobile.

OSS (COMPLEMENTO A DUE)

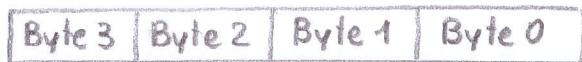
Complemento a due

REGISTRI

I registri sono delle memorie ad altissima velocità a bordo della CPU. In linguaggio assembly, sono identificati mediante dei nomi simbolici e possono essere usati in un programma come se fossero variabili.

L'IA32 ha 8 registri interi (A, B, C, D, DI, SI, SP, BP, IP) di dimensione 32 bit (4 Byte), di cui i primi sei possono essere usati come se fossero variabili per memorizzare INTERI e PUNTATORI. (Non dati in virgola mobile)

Rappresentiamo graficamente i registri nel seguente modo:



Byte più significativo

Byte meno significativo

Considerando singoli byte dei registri si ottengono sotto-registri.

REGISTRO	REGISTRO 4 BYTE (0-3)	SOTTOREGISTRO 2 BYTE (0-1)	SOTTOREGISTRO 1 BYTE (2-3)	SOTTOREGISTRO 1 BYTE (0-1)
General Porpose	A	ax	ah	al
	B	bx	bh	bl
	C	ch	cl	
	D	dx	dh	dl
	DI	di	/	/
	SI	si	/	/
	SP	sp	/	spl
	BP	bp	/	bpl

Il registro IP è l'INSTRUCTION POINTER ovvero quello contenente l'istruzione da eseguire, tale registro non può essere modificato direttamente.

OSS (Numero limitato Registri)

Si osservi che si hanno a disposizione un numero esiguo di registri, dunque durante la programmazione si dovranno riutilizzare i registri che non contengono dati utili in quella specifica sezione di codice.

OSS (Interferenza Dati)

Durante la programmazione occorre tenere a mente che se si modifica un sottoregistro, l'intero registro viene modificato. Ad esempio se salvo un dato in %eax e successivamente scrivo in %ah, il valore salvato in %eax sarà cambiato.

CONVENZIONI ABI

Vi sono diverse convenzioni dettate dall'Application Binary Interface (ABI) necessarie per il corretto funzionamento di moduli a basso livello, un esempio di questi è la convenzione per cui il valore di ritorno di una funzione deve essere memorizzato nel registro %eax. Un'altra convenzione è di usare principalmente i seguenti registri con questo ordine: A, C, D.

OSS

Queste convenzioni non sono legate all'ISA, e non sono nemmeno scelte dal costruttore ma da un consorzio di programmatore.

FUNZIONE ASH CON ARGOMENTI

Per poter scrivere una funzione in Assembly che abbia degli argomenti è necessario conoscere lo stato dello stack nel momento in cui la funzione viene chiamata.

Supponiamo che il main chiama una funzione f avendo due argomenti.

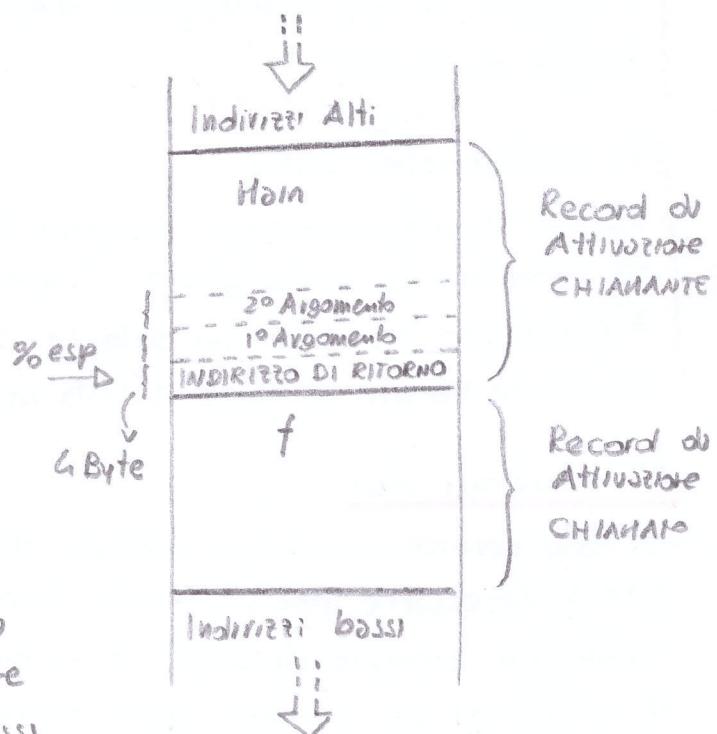
Al momento della creazione del RA di f, il registro %esp PUNTA

all'INDIRIZZO DI RITORNO, ovvero l'indirizzo della prossima istruzione che dovrà eseguire il main una volta terminata f, gli argomenti di quest'ultima sono memorizzati nel RA del main negli indirizzi successivi (maggiori) rispetto a quello puntato da %esp, si ricordi infatti che lo stack cresce verso indirizzi più bassi.

Per convenzione l'indirizzo di ritorno e gli argomenti sono memorizzati in 4 Byte, quindi per accedere al primo argomento la sintassi è $4(%esp)$ e per il secondo $8(%esp)$ e così via. Si noti che non si devono modificare i dati memorizzati in queste zone di memoria.

OSS (Differenza IA32 - IA64)

Nell'architettura IA32 i parametri delle funzioni sono memorizzati nello stack, nella IA64 invece sono salvati in specifici registri.



ESERCIZI

E1

```
// e1.c
int f()
return 5+7; → // e1.c
int f()
int a = 5;
a = a + 7;
return a;
```

```
// e1.s
.globl f → Esporta il simbolo "f"
f:
# a <-> eax
movl $5, %eax      # int a = 5;
addl $7, %eax      # a = a + 7;
ret                # return a;
```

OSS (movl su immediati)

Quando uso mov su immediati bisogno verificare se la costante può essere rappresentata con il numero di Byte rappresentato dal suffisso, in caso contrario viene troncato il valore dell'immediato.
In questo caso l'immediato è 5 rappresentabile con 4 Byte (1)

E2

```
// e2.c
int f()
return 3*4 + 2*5 - 1;
```

```
// e2.c.
int f()
int a = 3
a = a * 4
int c = 2
```

$c = c * 5$
 $a = a + c$
 $a--$
return a

// e2.s

```
.globl f
f:
# a <-> eax # c <-> ecx
movl $3, %eax
imull $4, %eax
movl $2, %ecx
imull $5, %ecx
addl %ecx, %eax
decl %eax
ret
```

Il primo passo è riscrivere il programma C in modo da fare una singola operazione per istruzione.

OSS (Ottimizzazione)

Tramite il comando `> gcc -m32 -S en.c`

possiamo generare il file ASH tuttavia noteremo che il compilatore avrà usato più istruzioni rispetto al nostro codice (oltre alle direttive `.directive`). Per ottenere un codice simile al nostro occorre aggiungere l'ottimizzazione nel comando gcc:

`> gcc -m32 -S en.c -O1`

Vi sono diversi livelli di ottimizzazione, il più basso è `-O1`, solitamente si usa `-O2`.

E3

```
// e3.c
int f(int x)
{
    return x+1
}

// e3.s
.globl f
f:
    movl 4(%esp), %eax
    incl eax
    ret
```

```
// e3.c
int f(int x)
{
    int a=x
    a++
    return a
}
```

E4

```
// e4.c
int somma(int x, int y)
{
    int a=x
    a=a+y
    return a
}
```

```
// e4.s
.globl somma
somma:
    movl 4(%esp), %eax
    addl 8(%esp), %eax
    ret
```

E5

```
// e5.c
int f(int x, int y, int z, int w)
{
    int a=x
    a=a*y
    c=z
    c=c*w
    d=a+c
    return d
}
```

```
// e5.s
.globl f
f:
    movl 4(%esp), %eax
    imull 8(%esp), %eax
    movl 12(%esp), %ecx
    imull 16(%esp), %ecx
    addl %ecx, %eax
    ret
```

Verifica

E6

```
1 // e6.c  
| int sgr(int x)  
|     int a=x;  
|     a=a*x  
|     ret
```

```
1 // e6.s  
| .global sgr  
| sgr:  
|     movl 4(%esp), %eax  
|     imull 4(%esp), %eax  
|     ret
```

5 HARZO

Ricordiamo che le istruzioni (opcode) sono divise in tre categorie:

- Movimento Dati (es: mov)
- Aritmetico-logiche (es: add, xor)
- Salto

Andiamo ad analizzare quest'ultima tipologia.

ISTRUZIONI DI SALTO

Normalmente, il flusso di controllo di un programma procede in modo sequenziale, eseguendo le istruzioni nell'ordine in cui appaiono in memoria. Ogni volta che un'istruzione "è" viene eseguita, il registro EIP (Instruction Pointer), che punta allo prossimo istruzione da eseguire, viene incrementato automaticamente del numero di byte occupati da "è".

Per alterare il flusso di controllo, ovvero modificare il contenuto di EIP si utilizzano un tipo di istruzioni chiamate ISTRUZIONI DI SALTO.

Vi sono tre tipi di istruzioni di SALTO:

- SALTO INCONDIZIONATO: Il registro EIP viene sovrascritto con l'indirizzo di memoria dell'istruzione a cui si vuole saltare.

- SALTO CONDIZIONATO: In questo caso il registro EIP viene sovrascritto solo se è verificata una determinata condizione sui dati.

- CHIAVATA E RITORNO DA FUNZIONE

• SALTO INCONDIZIONATO JMP

Tale istruzione ha la seguente sintassi:

Jmp label

dove "label" indica un punto del codice ASm.

es:

```
    movl $0, %eax  
L: incl %eax  
    jmp L  
          ↳ label (etichetta)
```

• SALTO CONDIZIONATO JCC

In questo caso si ha la seguente sintassi:

Jcc label

↳ Condition Code

L'istruzione JCC e' composta da un Condition Code (cc) che indica la CONDIZIONE da verificare. Se tale condizione risulta vera allora il programma effettua il salto a label.

Per comprendere come usare JCC occorre conoscere il REGISTRO EFLAGS

- Registro EFLAGS

Il registro EFLAGS e' a 32 bit in IA32 e alcuni sui bit rappresentano proprietà dell'ULTIMA operazione aritmetico-logica. (A-L)

I bit (FLAG) piu' importanti sono i seguenti:

• ZF (Zero Flag): Viene posto a 1 se l'ultima operazione A-L ha prodotto un valore zero, e 0 altrimenti.

• SF (Sign Flag): Viene posto a 1 se l'ultima operazione A-L ha prodotto un valore negativo, e 0 altrimenti.

• CF (Carry Flag): Viene posto a 1 se l'ultima operazione A-L ha generato un ripporto, e 0 altrimenti.

• OF (Overflow Flag): Viene posto a 1 se l'ultima operazione A-L ha generato un overflow, e 0 altrimenti.

I condition Code cc vanno a leggere le precedenti FLAGS contenute nel registro EFLAGS per valutare le condizioni.

OSS

Come vengono modificate le FLAG a seguito delle operazioni A-L viene descritto, senza specificare l'implementazione ma solo la logica, nel manuale Intel.

Si noti inoltre che l'operazione Mov non altera le flag.

- Condition Code (cc)

Vediamo ora le possibili condizioni su cui è possibile saltare ed i relativi suffissi. Si ricordi che la condizione si riferisce al risultato RIS dell'ultima operazione Arithmetico-Logica eseguita.

	SUFFISSO CC	SINONIMO	TEST SUL RISULTATO RIS
	e	z	RIS == 0
	ne	nz	RIS != 0
SOLO DATI SIGNED	g	nle	RIS > 0
	ge	nl	RIS ≥ 0
	l	nge	RIS < 0
	le	ng	RIS ≤ 0
	a	nbe	RIS > 0
SOLO DATI UNSIGNED	ae	nb	RIS ≥ 0
	b	nae	RIS < 0
	be	na	RIS ≤ 0

N.B. (DATI SIGNED - UNSIGNED)

La maggior parte delle operazioni di test precedenti richiedono la specifica se i dati su cui si sta lavorando sono con o senza Segno, il registro non memorizza alcuna informazione sul segno del dato memorizzato, sarà nostra premura saperlo. Naturalmente l'uso sbagliato del cc in relazione al segno porta ad una valutazione errata della condizione.

TRADUZIONE CONDIZIONI C

Il nostro obiettivo sarà quello di scrivere in ASH tutti i costrutti condizionali che troviamo in C (if, if-else, while, for, ecc...), per ciò è utile riscrivere il programma C negando la condizione e usando l'istruzione goto che permette di saltare ad un specifico punto del codice.

• ISTRUZIONE IF

Consideriamo il classico costrutto if in cui se una certa condizione "test" risulta vera viene eseguito un blocco di istruzioni "A" e infine, indipendentemente dalla condizione viene eseguito un blocco di chiusura "E" (Exit).

CODICE C	CODICE C EQUIVALENTE	TRADUZIONE IA32
if(test) A; E;	if(!test) goto E; A; E;	"Operazione per modifica EFLAGS" Jcc E A; E;

ESEMPIO

```
// abs.c
int abs(int x)
{
    if(x<0) return -x;
    return x;
```

Traduco quindi in ASH:

```
// abs.s
.global abs  # a<->eax
abs:
    movl 4(%esp), %eax
    subl $0, %eax  # eax = eax - 0
    jge E          # eseguo E se eax - 0 ≥ 0 ⇔ eax ≥ 0
    negl %eax     # inverto il segno di eax
E:
    ret
```

```
// abs_eq.c
int abs(int x)
{
    int a=x;
    if(a≥0) goto E;
    a=-a;
E:
    return a;
```

OSS (SIDE EFFECT SUI DATI)

Per poter valutare la precedente condizione abbiamo usato l'istruzione Sub che fa side-effect su D e su EFLAGS. In questo caso non abbiamo avuto problemi poiché di fatto D (%eax) rimane invariato ($D = D - 0$) in generale non è così, ovvero usando le istruzioni A-L standard potrei "sporcare" il dato da testare inoltre il risultato di tali istruzioni in questi casi NON ci interessa.

Per ovviare a questi problemi sono state introdotte specifiche istruzioni equivalenti che modificano EFLAGS ma non memorizzano il risultato in D, ovvero CHP e TEST

• ISTRUZIONI DI CONFRONTO CHP e TEST

Tali istruzioni sono pensate appositamente per eseguire confronti tra due operandi, CHP equivale a SUB, TEST ed AND, entrambi modificano il registro EFLAGS ma NON l'operando D.

- CHP

sub S,D		$D \leftarrow D - S$
cmp S,D		$D - S$

La seguente tabella riporta la condizione testata per ciascun prefisso assumendo di aver appena eseguito un'operazione Cmp S,D

SUFFISSO		CONDIZIONE TESTATA
SIGNED	UNSIGNED	
e	e	$D == S$
ne	ne	$D != S$
g	a	$D > S$
ge	ae	$D \geq S$
l	b	$D < S$
le	be	$D \leq S$

- TEST

and S,D | D < D & S

test S,D | D & S

L'istruzione TEST viene usata per verificare se un registro Reg è nullo o meno, infatti ponendo $S = D = \text{Reg}$ si ha la seguente equivalenza:

$$\text{testl Reg, Reg} \Leftrightarrow \text{cmpl } \$0, \text{Reg} \rightarrow (\text{Reg} = 0)$$

Si capisce il motivo dell'equivalenza dal seguente esempio

es:

$$- \%al = 10010010$$

$$\%al \& \%al = \%al \neq 0$$

$$- \%al = 00000000$$

$$\%al \& \%al = \%al = 0$$

$$\begin{array}{r}
 10010010 \\
 10010010 \\
 \hline
 10010010
 \end{array}$$

$$\begin{array}{r}
 00000000 \\
 00000000 \\
 \hline
 00000000
 \end{array}$$

• ISTRUZIONE IF-ELSE

Nel caso di if-else, se la condizione "test" risulta vera viene eseguito un blocco "A", altrimenti un blocco "B", ed alla fine sempre il blocco "E" (Exit).

CODICE C	CODICE C EQUIVALENTE	TRADUZIONE IA32
<pre>if(test) A; else B; E;</pre>	<pre>if(!test) goto B; A; goto E; B; E;</pre>	<p>"Operazione per EFLAGS"</p> <p>Jcc B A Jmp E B E</p>

• CICLO WHILE

Se la condizione "test" è verificata esegue in loop il blocco A, alla fine del ciclo esegue il blocco E.

CODICE C	CODICE C EQUIV.	TRADUZIONE IA32
while(test) A; E;	L: if(!test) goto E; A; goto L; E;	L: "Operazione per EFLAGS" JCC E A JMP L E

• CICLO FOR

Consideriamo un ciclo for in cui per inizializzare i dati usiamo un blocco "I", se verificata la condizione "test" continua il ciclo, ed ad ogni iterazione esegue un blocco "A" ed il blocco "P".

Al termine del ciclo si esegue E.

CODICE C	CODICE C EQUIVALENTE	TRADUZIONE IA32
for(I; test; P) A; E;	I; L: if(!test) goto E; A; P; goto L; E;	I L: "Operazione EFLAGS" JCC E A P JMP L E

RAPPRESENTAZIONE DEI NUMERI IN MEMORIA

L'ENDIANESS di un processore definisce l'ordine con cui vengono disposti in memoria i byte della rappresentazione di un valore numerico memorizzato ad esempio in un registro.

Supponiamo che nel registro %eax si memorizzi il valore esadecimale 0xDEADBEEF: (`movl $0xDEADBEEF, %eax`), ed esaminiamo i due tipi di rappresentazione.

$$A = \%eax$$

DE	AD	BE	EF
Byte 3	Byte 2	Byte 1	Byte 0

• BIG-ENDIAN

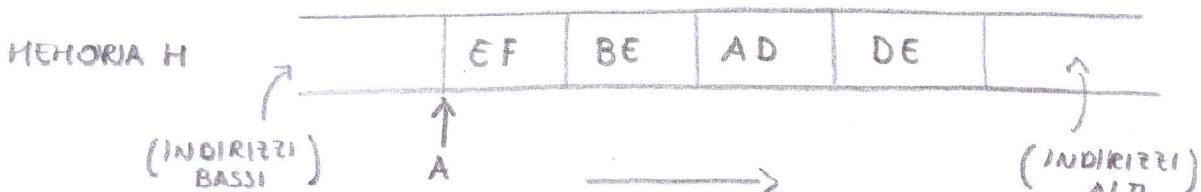
In questa rappresentazione il byte più significativo del numero A viene posto all'indirizzo più basso, ovvero il numero è copiato byte-bytes nell'ordine corretto.



Tale rappresentazione è usata nei processori PowerPC e SPARC.

• LITTLE-ENDIAN

In questo caso il byte meno significativo del numero A viene posto all'indirizzo più basso, ovvero il numero è copiato byte-bytes invertendo l'ordine.



Tale rappresentazione è usata nei processori della famiglia x86.

PUNTATORI IN ASSEMBLY

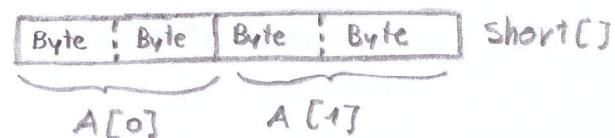
- In Assembly gestiamo i puntatori tramite i seguenti OPERANDI A MEMORIA.

SINTASSI	VALORE DENOTATO	NOME CONVENZIONALE
(Reg)	$M[R[\text{Reg}]]$	Indirizzamento indiretto a registro
$d(\text{Reg})$ Spostamento	$M[d + R[\text{Reg}]]$	Indirizzamento indiretto a registro con spostamento
$(\text{Reg}1, \text{Reg}2)$ Base \downarrow Indice	$M[R[\text{Reg}1] + R[\text{Reg}2]]$	Indirizzamento indiretto a registro con base e indice
$(\text{Reg}1, \text{Reg}2, S)$ Scala \downarrow (1, 2, 4)	$M[R[\text{Reg}1] + R[\text{Reg}2] \cdot S]$	Indirizzamento indiretto a registro con base, indice e scala
$d(\text{Reg}1, \text{Reg}2, S)$	$M[d + R[\text{Reg}1] + R[\text{Reg}2] \cdot S]$	Indirizzamento indiretto a registro con base, indice, scala e spostamento

OSS (Operando (Reg1, Reg2, S))

Tale operando è utile per leggere gli elementi di un array.
 L'argomento s (scala) serve per scandire l'array in blocchi di byte,
 ad esempio in un array A di short (2 byte per elemento) userò
 la scala $s=2$.

- | |
|--|
| $\left\{ \begin{array}{l} s=1 \rightarrow \text{Array di (unsigned) char} \\ s=2 \rightarrow \text{Array di (unsigned) short} \\ s=4 \rightarrow \text{Array di (unsigned) int} \end{array} \right.$ |
|--|



ESERCIZI 190306

E1

```

// e1.c
// Raddoppia il contenuto puntato
void times2(short* p)
{
    *p = *p * 2;
}

// e1.s
.globl times2
times2:
    movl 4(%esp), %ecx    # Salvo il puntatore p in ecx (p = 4 Byte)
    movw (%ecx), %ax      # Copio il contenuto di p in ax
    addw %ax, (%ecx)      # Somma al contenuto di p il valore ax (= *p)
    # Uso la somma poche' meno costosa rispetto
    # alla moltiplicazione
    ret                   # Anche se la funzione e' void

```

N.B. (Dimensioni Puntatori)

In IA32 TUTTI i puntatori, di qualsiasi tipo, hanno dimensione di 4 Byte (32 bit).

E2

```

// e2.c
void swap(char*x, char*y)
{
    char tmp = *x
    *x = *y
    *y = tmp;
}

// e2.s
.globl swap
swap:
    movl 4(%esp), %ecx
    movl 8(%esp), %edx } →
    movb (%ecx), %al
    movb (%edx), %ah
    movb %ah, (%ecx)
    movb %al, (%edx)
    ret

```

```

// e2_eq.c
void swap(char*x, char*y)          x → █     y → █
{
    char* c = x           x → █ VAL1 ← c
    char* d = y           y → █ VAL2 ← d
    char al = *c          // al = VAL1 (Salvo i valori)
    char ah = *d          // ah = VAL2
    *c = ah              // VAL1 = VAL2 (Swap)
    *d = al              // VAL2 = VAL1

```

N.B. (Promozione parametri C)

Anche se gli argomenti di swap() sono char (1 Byte), nello stack vengono sempre memorizzati su 4 Byte.

E3

```
// e3.c (swap su short)
void swap (short* x, short* y)
short tmp = *x
*x = *y
*y = tmp

// e3.s
.globl swap
swap:
    movl 4(%esp), %ecx
    movl 8(%esp), %edx
    movw (%ecx), %cx
    movw (%edx), %dx
    movl 4(%esp), %eax
    movw %dx, (%eax)
    movl 8(%esp), %eax
    movw %cx, (%eax)
ret
```

```
// e3_eq.c
void swap (short* x, short* y)
short* c = x      x → [v1] ← c
short* d = y      y → [v2] ← d
short cx = *c      // cx = v1
short dx = *d      // dx = v2
short* a = x      a → [v1]
*a = dx          // v1 = v2
*a = cx          a → [v2]
// v2 = v1
return
```

E4

```
// e4.c (lettura elementi array)
short get (short* v, unsigned n, unsigned i)
if (i >= n) return -1
return v[i]
```

```
// e4.s
.globl get
get: movl 4(%esp), %ecx
    movl 8(%esp), %edx
    movl 12(%esp), %eax
    cmpl %edx, %eax
    jbe E
    movw (%ecx, %eax, 2), %ax
ret

E:
    movw $-1, %ax
ret
```

```
// e4_eq.c
short get (short* v, uns. n, uns i)
short* c = v
unsigned d = n
unsigned a = i
if (a >= d) goto E;
short ax = *(c+a); // c[a]
return ax

E:
    ax = -1
    return ax
```

E5

```

| // e5.c
| short sum(short* v, int n)
|     int i;
|     short a = 0;
|     for(i=0; i<n; i++)
|         a=a+v[i];
|     return a;
|
| // e5.s
| .globl sum
| sum: movl 4(%esp), %ecx
|       movl 8(%esp), %edx
|       xorw %edx, %edx
|       decl %edx
| L:  cmpl $0, %edx    # equivalent testl %edx, %edx
|      jne E
|      addw (%ecx, %edx, 2), %eax
|      decl %edx
|      jmp L
| E: ret

```

```

| // e5-eq.c
| short sum(short* v, int n)
|     short* c=v;
|     int d = n; // Ho rimosso la variabile i
|     short a = 0; d--;
|     L: if(d<a) goto E;
|         a=a+*(c+d);
|         d--;
|         goto L;
| E: return a;

```

E6

```

| // E6.C
| unsigned my_strlen(const char* s)
|     unsigned cnt=0;
|     while(*s++) cnt++;
|     return cnt
|
| 
```

```

| // e6-eq.c
| 
```

E6 + N.B.

E7

SUDDIVISIONE REGISTRI

Abbiamo già visto in precedenza che a disposizione del programmatore in IA32 ci sono sei registri General Purpose, tuttavia ci siamo limitati all'uso di solo tre di questi: A, C, D, ciò è dovuto ad una classificazione dei registri stabilita dal ABI (Application Binary Interface) secondo cui ci sono due tipi di registri general purpose:

- **REGISTRI CALLER-SAVE (A, C, D)**: Quando una funzione f() chiama un'altra funzione g(), quest'ultima può utilizzare liberamente questi registri, il chiamante f() se interessato ai dati memorizzati in A, C, D deve provvedere al loro salvataggio prima di chiamare g(). poiché quest'ultima potrebbe averne variato il contenuto.
Sono soprannominati dunque CALLER-SAVE dato che è il caller (chiamante) a memorizzare i dati in essi contenuti, se di suo interesse, il callee (chiamato) invece può "sporcarli" liberamente.
- **REGISTRI CALLEE-SAVE (B, DI, SI)**: Quando una funzione f() chiama una funzione g(), al termine di quest'ultima, troverà sempre INALTERATI i dati in questi registri, poiché il callee g() (chiamato) se li utilizza, deve PRIMA memorizzare i valori iniziali in modo da ri-pristinarli una volta terminata.
Questi registri sono quindi utilizzabili liberamente dal chiamante poiché le funzioni che andrà a chiamare avranno cura di mantenere i dati in essi contenuti.
Altri registri callee-save sono SP e BP ma essi NON sono general purpose ma usati nella gestione dello stack dei record di attivazione.

OSS (Function Private)

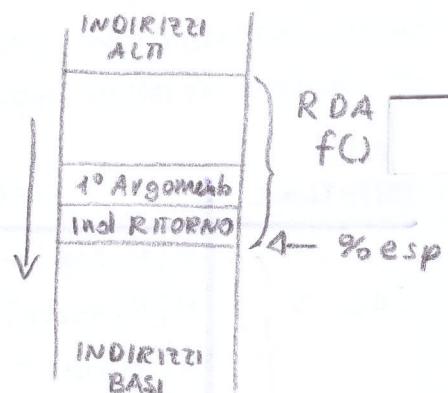
Tali convenzioni sono FONDAMENTALI per il corretto funzionamento di più moduli (file) scritti da diversi programmati. Tuttavia internamente al modulo possono essere violate a patto che però siano rispettate per l'interfacciamento esterno al modulo.

MOVIMENTO DATI DA/VERSO LA STACK PER CHIAMATE

Il nostro obiettivo è quello di scrivere funzioni in ASH che nel loro codice eseguano chiamate ad altre funzioni, ad esempio la funzione $f()$ chiama una funzione $g(\text{int } x)$.

Abbiamo già visto come deve apparire la stack al momento dell'inizio di $g(x)$:

- I byte sopra quelli puntati da $\%esp$ devono contenere gli argomenti di $g(\text{int } x)$
- Il registro $\%esp$ deve puntare all'indirizzo di ritorno di $f()$.



• SCRITTURA ARGOMENTI

Per poter effettuare questa operazione occorre allocare memoria nella stack, per farlo è sufficiente spostare verso il basso il puntatore $\%esp$ e memorizzare i dati nello spazio allocato. A questo scopo esiste l'istruzione **PUSH** e in contrapposizione ad essa **POP** usata per deallocare memoria dalla stack (sposta $\%esp$ verso l'alto).

ISTRUZIONE	EFFETTO	DESCRIZIONE
pushl s	$R[\%esp] \leftarrow R[\%esp] - 4$ $H[R[\%esp]] \leftarrow s$	Copia l'operando s di 4 Byte sulla cima della stack
popl d	$D \leftarrow H[R[\%esp]]$ $R[\%esp] \leftarrow R[\%esp] + 4$	Toglie i 4 Byte dalla cima della stack e li copia in D

Dunque la PUSH si occupa di spostare $\%esp$ e memorizzare nello nuovo zona allocata l'operando D , tale istruzione la useremo per inserire gli argomenti, l'indirizzo di ritorno invece sarà inserito da CALL che vedremo dopo.

DSS (Dimensione 4 Byte Stack)

La stack è composta da slot di dimensione costante di 4 Byte, quindi le istruzioni PUSH e POP hanno sempre suffisso "l".

SCRITTURA INDIRIZZO DI RITORNO

Una volta inseriti gli argomenti della funzione da chiamare occorre far puntare %esp all'indirizzo dell'istruzione di ritorno e quindi effettuare la chiamata alla funzione ($f \rightarrow g(intx)$), entrambe le operazioni sono svolte dall'istruzione CALL. Tale istruzione insieme a RET appartengono ad un tipo di istruzioni relativo appunto alle chiamate e ritorno da funzione.

ISTRUZIONE	EFFETTO	DESCRIZIONE
CALL S	$R[\%esp] \leftarrow R[\%esp] - 4$ $H[R[\%esp]] \leftarrow R[\%eip]$ $R[\%eip] \leftarrow S$	CHIAMATA A FUNZIONE: mette in stack l'indirizzo dell'istruzione successiva alla CALL (indirizzo di ritorno) e salta all'indirizzo specificato dall'operando S
RET	$R[\%eip] \leftarrow H[R[\%esp]]$ $R[\%esp] \leftarrow R[\%esp] + 4$	RITORNO DA FUNZIONE: toglie dalla stack l'indirizzo di ritorno e lo scrive in EIP

Esempio p 29

andare in stack ed BEIP

PROLOGO ED EPILOGO

Quando scriviamo una funzione $f()$ che nel suo corpo conterrà una chiamata ad una funzione $g()$ è conveniente utilizzare come registri di lavoro B , DI , SI poiché per convenzione i dati in essi contenuti non saranno sovrascritti dalla funzione $g()$, o meglio, tali registri possono essere usati da $g()$, ma prima che termini deve rispristinarli ai valori iniziali, dunque dal punto di vista di $f()$ essi rimangono intatti.

Questa operazione di preservazione dei dati memorizzati in B , DI , SI (callee-save) spetta anche alla funzione $f()$ dato che essa stessa viene invocata da altre funzioni come il main che si aspettano di trovare i dati invariati. La funzione $f()$ inoltre deve provvedere anche a caricare gli argomenti di $g()$ nello stack e una volta eseguita $g()$ deve rimuoverli.

Le istruzioni atte a fare queste azioni le mettiamo in due blocchi, uno ad inizio codice (PROLOGO) e l'altro appena prima della RET (EPILOGO)

• PROLOGO

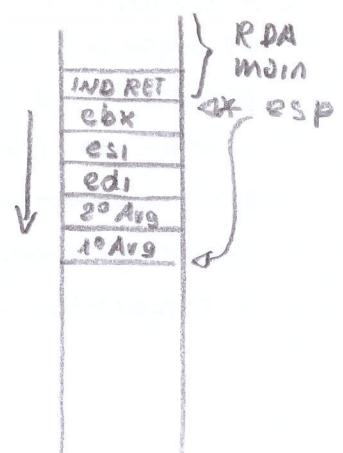
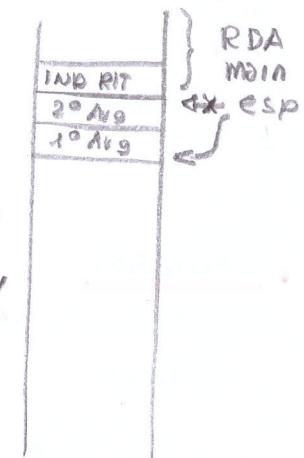
Prima di tutto se utilizziamo uno dei registri callee-save doppiamo memorizzate i valori nella stack tramite PUSH, poi di seguito inserire gli argomenti di $g()$. Per quest'ultima operazione invece di usare la PUSH allochiamo memoria nello stack (spostando $%esp$) in modo da poter contenere gli argomenti della funzione chiamata (da $f()$) che ha più argomenti di tutte le altre, in questo modo all'interno del codice di seguito non varierà la posizione di $%esp$. (nell'immagine supponiamo che la funzione chiamata da $f()$ con il massimo numero di argomenti, ne abbia due).

A questo punto nel codice successivo quando invocheremo una funzione salveremo sempre i suoi argomenti in $(%esp)$, $4(%esp)$ ecc.

PROLOGO {
pushl %ebx
pushl %esi
pushl %edi
subl \$8, %esp # Alloc 2-4 Byte
:

movl 1°Arg, (%esp)
movl 2°Arg, 4(%esp)
call g
:

EPILOGO {
...
ret



• EPILOGO

In questo blocco occorre fare le operazioni opposte a quelle fatte nel prologo e in ordine inverso in modo da ripristinare lo stack come ad inizio esecuzione di `f()` e ripristinare i registri calle-savve.

Per deallocare la memoria usata per gli argomenti è sufficiente usare una ADD, e per i registri calle-savve usare la POP in ordine opposto alla PUSH.

Si ricorda che lo stack è una pila (Last Input First Output) (LIFO) e che cresce verso indirizzi BASSI

La struttura del nostro codice sarà quindi la seguente:

PROLOGO	<code>pushl %ebx</code> <code>pushl %esi</code> <code>pushl %edi</code> <code>subl \$K, %esp</code> <code>:</code> <code>addl \$K, %esp</code> <code>popl %edi</code> <code>popl %esi</code> <code>popl %ebx</code>
---------	---

dove:

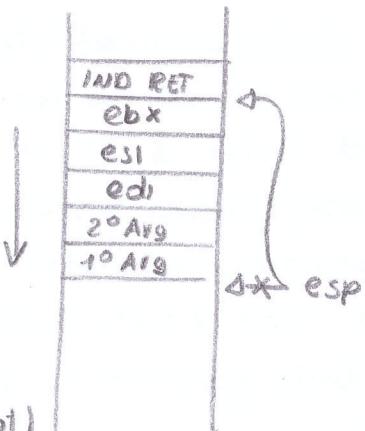
$K = \text{Numero di Byte necessari a rappresentare gli argomenti della funzione da chiamare con più argomenti}$

• VANTAGGI

Utilizzare i registri calle-savve ed adottare questa struttura ha diversi vantaggi:

- UN SOLO SALVATAGGIO E RIPRISTINO: Se invece di usare B, D1, S1 usassimo A, C, D, prima di ogni chiamata a funzione dovremmo salvare i dati e poi ripristinarli. In questo modo il salvataggio di B, D1, S1 avviene nel PROLOGO ed il ripristino nell'EPILOGO una sola volta.

- ARGOMENTI IN POSIZIONE COSTANTE: Una volta allocata la memoria per gli argomenti della funzione con più argomenti, useremo tale zona per passare gli argomenti anche alle altre funzioni e li scriveremo sempre in (%esp), 4(%esp), 8(%esp) ecc. Non seguendo questo metodo ad ogni CALL lo "spostamento" da %esp in cui salvare gli argomenti sarebbe diverso, e ciò porterebbe facilmente a commettere errori di clistrazione.



UPCASTING E DOWNCASTING

Al momento del passaggio di parametri ad una funzione, il compilatore esegue l' INTEGER PROMOTION, ovvero indipendentemente dalla taglia dell'argomento quest'ultimo viene esteso a 4 Byte (UPCASTING), e memorizzato sulla stack.

• UPCASTING

Il processo di UPCASTING deve preservare il valore iniziale rappresentato nella taglia più piccola, cioè la variabile estesa deve avere il medesimo valore di quella iniziale.

Tale processo dipende se il dato è con segno oppure senza segno.

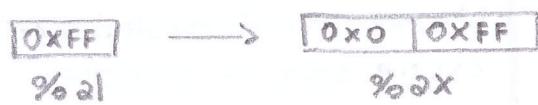
- Upcasting su unsigned

In questo caso è sufficiente riempire i byte di differenza con 0.

es:

8 bit → 16 bit

| 0x0 = 0000 0000
| 0xFF = 1111 1111



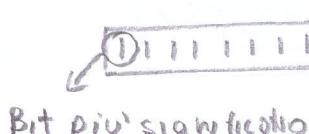
- Upcasting su signed

Poiché i dati di tipo signed sono rappresentati in complemento a due non si può riempire semplicemente con 0 poiché si andrebbe a cambiare il valore rappresentato. Per effettuare upcasting su dati di tipo signed occorre riempire i bit di differenza con il bit

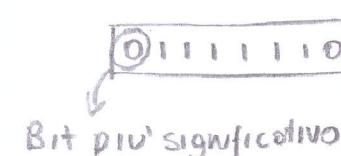
più significativo del dato iniziale. (taglia minore)

es:

8 bit → 16 bit



Bit più significativo



Bit più significativo

• ISTRUZIONI ASH PER UPCASTING

Dato che i nostri moduli ASH si interfacciano con moduli C e' necessario rispettare la convenzione del integer promotion. Per automatizzare le operazioni descritte in precedenza esistono le seguenti istruzioni:

ISTRUZIONE	EFFETTO	DESCRIZIONE
Hovz S,D	$D \leftarrow \text{ZeroExtend}(S)$	Esegue l'upcasting di S (unsigned) e lo memorizza in D (Dim S > Dim D)
Hovs S,D	$D \leftarrow \text{SignedExtend}(S)$	Esegue l'upcasting di S (signed) e lo memorizza in D (Dim S > Dim D)

Le precedenti istruzioni necessitano di due suffissi, il primo specifica la taglia della sorgente S ed il secondo quella della destinazione D.

Movz HH
Movsh H
Dim_S → Dim D

N.B. (VINCAO Mov)

L'operando di destinazione D può essere solo un REGISTRO

• DOWN CASTING

L'operazione di DOWNCASTING e' l'opposto dell'upcasting, ovvero corrisponde a troncare una variabile.

In ASH si effettua con la MOV utilizzando il suffisso opportuno che indica il numero di Byte spostati e quindi la taglia della variabile che contiene il dato troncato.

es:

16 bit → 8 bit

0xAB	0xCA	→	0xCA	movb %ax,%cl
%ax			%cl	

Si ricorda che il valore 0xCA e' già accessibile in %al

VARIABILI LOCALI

Finora abbiamo memorizzato le variabili, che usiamo nelle funzioni, direttamente nei registri della CPU, tuttavia non e' sempre possibile. Gli scenari piu' comuni che non permettono l'uso dei registri sono i seguenti:

- Non si hanno piu' registri disponibili.
- Memorizzazione di Array o Struct
- Utilizzo operando & (address of): Se vogliamo che una funzione g() faccia side-effect su variabili locali di f() e' necessario che queste siano memorizzate nella stack, in particolare nel record di attivazione RDA di f().

es:

```
int f()
{
    int x;
    g(&x); // Non posso memorizzare x in un registro
```

OSS (Limitatezza Stack)

Nella stack e' possibile memorizzare una quantita' limitata di dati, se si necessita di molto spazio (es 1GB) si utilizza l'heap.

INDIRIZZAMENTO VARIABILI IN ASH (LEA) (Load Effective Address)

L'istruzione LEA consente di sfruttare la flessibilita' data dai modi di indirizzamento a memoria per calcolare espressioni aritmetiche che coinvolgono somme e prodotti su indirizzi di interi.

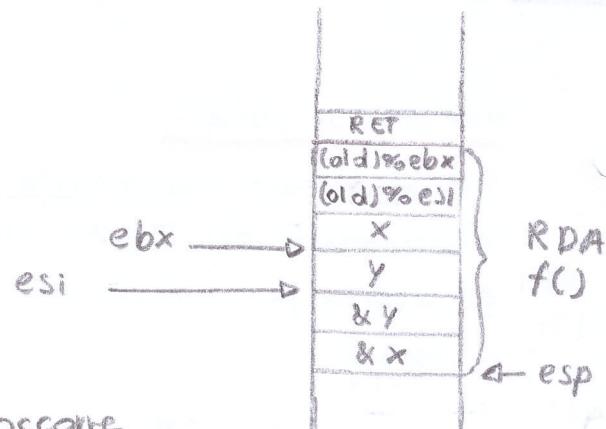
ISTRUZIONE	EFFETTO	DESCRIZIONE
leal S, D _b	D _b ← &S	Calcola l'indirizzo effettivo specificato dall'operando di tipo memora S e lo scrive in D.

OSS

Si osservi che leal, diversamente da movl, non effettua un accesso a memoria sull'operando sorgente. L'istruzione leal calcola infatti l'indirizzo effettivo dell'operando sorgente, senza perciò accedere in memoria a quell'indirizzo.

ESEMPIO

```
| int f()
| {
|     int x, y
|     g(&x, &y)
|     return x+y
| }
```



Dato che g() fa side effect su x e y occorre memorizzarle nello stack, e per evitare di calcolarne più volte gli indirizzi li memorizzano in %ebx e %esi

Supponendo la gestione dello stack come descritto in figura, il codice ASH corrispondente è il seguente:

```
.globl f
f:
| pushl %ebx
| pushl %esi
| subl $16, %esp } PROLOGO
|
| leal 12(%esp), %ebx → { movl %esp, %ebx
| leal 8(%esp), %esi } addl $12, %ebx
|
| movl %ebx, (%esp) # 1° Arg (&x)
| movl %esi, 4(%esp) # 2° Arg (&y)
|
| call g
|
| movl (%ebx), %eax
| addl (%esi), %eax
|
| addl $16, %esp } EPILOGO
| popl %esi
| popl %ebx
|
| ret
```

OSS (Passaggio per copia)

Si potrebbe pensare di strutturare diversamente le funzioni f() e g() ovvero passando a g() direttamente le variabili locali di f(). Ciò non è consentito poiché in C gli argomenti passati ad una funzione devono essere una copia, infatti in generale la funzione chiamata potrebbe modificare gli argomenti che gli sono stati passati.

• ESPRESSIONI ARITMETICHE CON ISTRUZIONE LEA

L'istruzione LEA può essere adottata per il calcolo di espressioni aritmetiche, sfruttando il fatto che questo non effettua accesso in memoria, ma calcola un indirizzo specificato dall'operando Memoria S e lo memorizza in D, dunque esegue solo un operatore aritmetico indipendentemente dal fatto che S esprima o meno un indirizzo di memoria.

L'operando memoria più generico è il seguente:

$$d(\text{Reg}_1, \text{Reg}_2, S)$$

/

1, 2, 4, 8

dove:

d = immediato (offset)

Reg₁ = Base

Reg₂ = Indice

S = spostamento (1, 2, 4, 8)

tale operando corrisponde alla
seguente espressione aritmetica:

$$d(\text{Reg}_1, \text{Reg}_2, S) = d + \text{Reg}_1 + (\text{Reg}_2 \cdot S)$$

es:

Per calcolare l'espressione: $-7 + 15 + 20 \cdot 2$

e memorizzarne il risultato nel registro D, esegue le seguenti istruzioni:

```
; movl $15, %eax
; movl $20, %ecx
; leal -7(%eax, %ecx, 2), %edx
```

N.B. (LEA NON VARIA EFLAGS)

L'istruzione Mov e Lea non influenzano le flags del registro EFLAGS, dunque i risultati delle espressioni aritmetiche calcolate mediante LEA non possono essere usate per l'attivazione dei Condition Code (cc)

ISTRUZIONE SETcc

L'istruzione SETcc permette di assegnare i valori 0 o 1 ad un registro a 8 bit o ad un byte di memoria a seconda della verifica o meno della condizione specificata dal Condition Code (cc).

ISTRUZIONE	EFFETTO	DESCRIZIONE
SETcc D _i	D _i ← Condizione	Se la condizione associata al suffisso cc è verificata, scrive 1 in D _i , altrimenti 0.

OSS

L'istruzione SETcc lavora solo con destinazioni 1 Byte ed inoltre se verificato il cc, memorizza il binario 1 = 0000 0001 altrimenti 0 = 0000 0000.

Tale istruzione è preferibile in sostituzione alle istruzioni di salto condizionato Jcc, tuttavia non sono sempre sostituibili. Si ricorda infatti che nelle condizioni C esiste la CORRO-CIRCUITAZIONE delle espressioni logiche && e ||, e molto spesso i programmati la sfruttano, ovvero exp₂ può essere calcolata se e solo se exp₁ risulta valido.
es:

if(exp₁ && exp₂) → Se exp₁ non è verificato, allora exp₂ non viene calcolato

if(exp₁ || exp₂) → Se exp₁ è verificato, allora exp₂ non viene calcolato.

Nel caso in cui exp₂ è calcolabile senza che abbia effetto sul programma, è possibile sostituire, in ASH, le condizioni composte in C tramite le istruzioni AND oppure OR e valutare l'intera condizione tramite SETcc. (Vedremo un esempio negli esercizi)

RIEPILOGO ORGANIZZAZIONE STACK FRAME

Nel caso piu' generale, ovvero considerando una chiamata a funzione e l'uso di variabili locali, il record di attivazione della nostra funzione e' diviso in tre parti:

1. BACKUP DI REGISTRI CALLEE-SAVE: Tramite la PUSH

mi memorizza i valori iniziali dei registri callee-save in modo da poterli ripristinarli prima di tornare al mio chiamante. (RET)

2. VARIABILI LOCALI: Sono quelle variabili che non

posso memorizzare nei registri. (es array/struct)

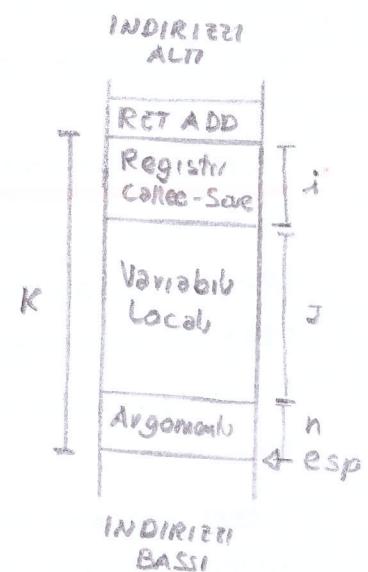
Oppure di cui devo conoscere l'indirizzo (& address-of)

3. NUMERO MASSIMO DI ARGOMENTI: Considerando la

funzione chiamata con piu' argomenti, mi abbozzi memoria sufficiente. Per ogni chiamata a funzione usero' sempre la seguente zona di memoria per il passaggio di parametri.

CONVENZIONE ABI (Application Binary Interface)

Una convenzione del ABI e' che se esp deve essere sempre un multiplo di 4, ovvero il mio record di attivazione deve avere dimensione k multiplo di 4. La zona di backup dei registri e' gia' multiplo di 4 dato che la PUSH lavora solo a 4 Byte, e lo e' anche la zona per gli argomenti dato che per l'integer promotion di C qualsiasi tipo di parametri e' passato su 4 Byte. Dunque dovremo fare attenzione solo alla zona per le variabili locali ed assicurarsi che abbia dimensione multiplo di 4, avvolgendo per eccesso il numero di byte necessari, aggiungendo dunque byte "inutili" detti di PADDING.



$$K = i + j + n$$

↓ ↓ ↗
 Registri Variabili Numero massimo
 Callee-Save Locali Argomenti
 (Multiplo di) (Dove assicurarmi
 4 che sia un multiplo
 di 4)

OSS (ABI MODERNE)

L'attuale ABI a 32 bit tuttavia impone che %esp sia multiplo di 16 e non di 4, in questo corso useremo la convenzione a 4, ma occorre tenere presente che le funzioni che si basano sull'ABI moderna potrebbero non funzionare correttamente su %esp multiplo di 4.

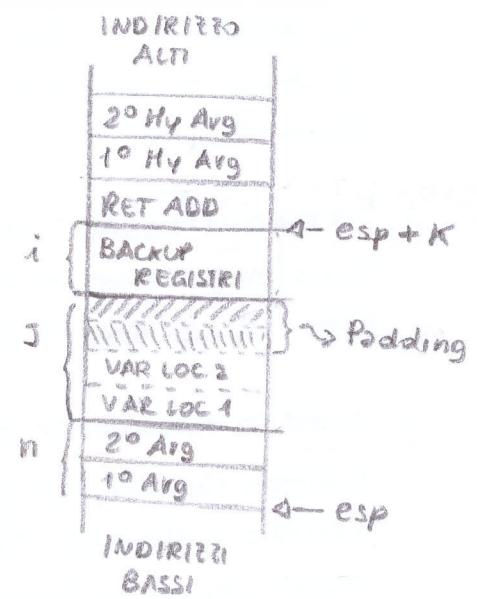
• POSIZIONE ARGOMENTI

Nei primi programmi ASH i nostri argomenti erano sempre disponibili a partire da 4(%esp), nel caso di allocazione di memoria nel stack ciò non risulta più vero dato che abbiamo spostato %esp.

I miei argomenti e quelli delle funzioni da chiamare avranno le seguenti posizioni:

- 1° Hy Arg 4 + esp + k
- 2° Hy Arg 8 + esp + k
- 1° Arg 0 + esp
- 2° Arg 4 + esp

doce:
$$k = i + j + n$$



OSS (PADDING)

Si noti nella figura, nell'aver supposto che la mia funzione abbia due variabili locali di tipo char, siamo stati costretti ad aggiungere due byte di padding.

VINCOLI VARIABILI LOCALI

Oltre al vincolo sulla dimensione del blocco di memoria per le variabili locali nella stack (multiplo di quattro), ci sono altre convenzioni ABI relative al tipo di dato memorizzato che impongono una certa disposizione o l'aggiunta di padding.

• VINCOLO SU VARIABILI SCALARI

Per variabili scalari intendiamo i soliti tipi (char, int, int* ecc).

Su tali variabili deve essere rispettata la seguente regola:

Un oggetto di tipo T deve essere allineato ad un indirizzo
MULTIPIO di `sizeof(T)`

- esempio 1

Consideriamo le seguenti variabili locali: (scalari)

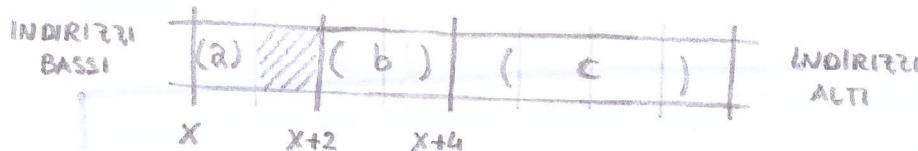
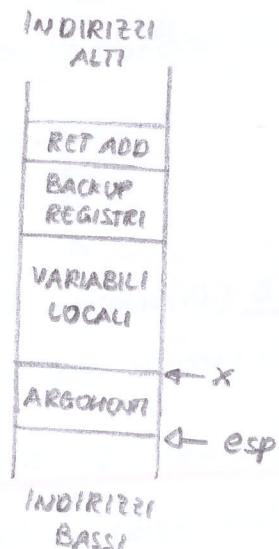
char a // `sizeof()` = 1

short b // `sizeof()` = 2

int c // `sizeof()` = 4

Nel memorizzarle nel blocco di memoria puntato da X occorre osservare la precedente regola.

Se li memorizziamo nell'ordine in cui sono dichiarate dovremo aggiungere byte di padding.

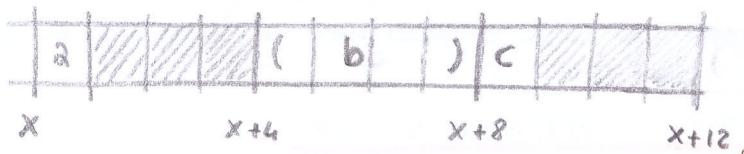


Dato che %esp deve essere multiplo di 4, si ha necessariamente che X e' un indirizzo pari, dunque X+2 e' multiplo di `sizeof(short) = 2`. La variabile short b NON può essere memorizzata a partire da X+1 poiché tale indirizzo NON e' multiplo di due, dobbiamo quindi inserire un byte di padding.

- esempio 2

Tenendo presente la stessa struttura per lo stack, andiamo a memorizzare i seguenti scalari.

char a // 1
int b // 4
char c // 1



Si osservi che dopo la variabile c sono stati aggiunti 3 byte di padding per rispettare la regola di % esp multiplo di 4.

Da questo esempio notiamo che l'ordine in cui memorizziamo le variabili locali incide sulla spazio occupa complessivamente.

Un metodo per ridurre tale spazio e' quello di inserire prima i dati con dimensione maggiore e poi quelli con dimensione minore (int \rightarrow char). (tale metodo e' usato dal compilatore.)



OSS

Con tale metodo ho ridotto notevolmente lo spazio occupato ($12 \rightarrow 8$)

N.B. (Dimensione Puntatori)

Si ricorda che TUTTI i tipi di puntatori hanno la medesima dimensione: 4 Byte

• VARIABILI DI TIPO ARRAY

Nella memorizzazione nello stack di un array occorre seguire la seguente regola:

Un oggetto di tipo ARRAY di T deve essere allineato ad un indirizzo multiplo di sizeof(T).

OSS

Tale regola deriva dalla precedente del caso scalare, infatti se l'array inizia con un indirizzo che e' multiplo di sizeof(T) allora tutti gli elementi che lo compongono rispetteranno la regola scalare.

• VARIABILI DI TIPO STRUCT

In questo caso devo rispettare tre regole che si basano essenzialmente su quelle viste per scalari e vettori.

a - TUTTI I CAMPI ALLINEATI: I campi che compongono la struct devono rispettare la regola valida per gli scalari.

b - INIZIO STRUCT: Sia T il tipo del campo con dimensione maggiore, allora l'indirizzo della struct deve essere un multiplo di `sizeof(T)`.

Tale regola garantisce che la variabile (campo) di tipo T sia allineata, ovvero rispetti la regola per gli scalari.

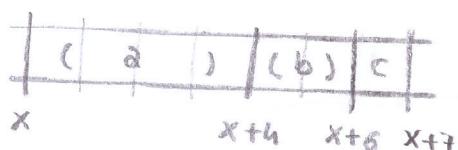
c - PADDING PER ARRAY DI STRUCT: La dimensione della struct deve essere un multiplo di `sizeof(T)`, dove T è il tipo del campo con dimensione maggiore.

Tale regola consente di far rispettare a tutti gli elementi di un array di struct la regola b.

- Esempio

Consideriamo la seguente struct s

Applicando la regola a, la mia struct viene memorizzata in memoria nel seguente modo:

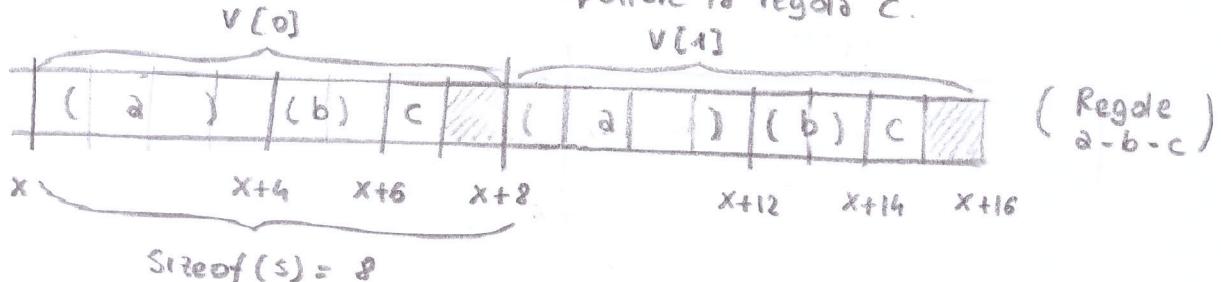


(Solo Regola a e b)

```
; typedef struct s {  
;     int a  
;     short b  
;     char c  
;} s
```

Se però considero un array di struct (struct consecutive) mi accorgo che la struct successiva inizierebbe a $x+7$ e ciò andrebbe a violare la regola a - b per il secondo elemento. Si aggiungono quindi padding alla fine della struct in modo da rispettare la regola c.

s v[2]



In codice ASH per accedere ai singoli campi della struct usa la solita aritmetica dei puntatori ricordando tutte le regole di padding.

Nel nostro esempio, considerando l'indirizzo della struct come "base" i singoli campi si ottengono con:

int a //4 Posizione: 0 + Base (Base)

short b //2 Posizione: 4 + Base 4 (Base)

char c //1 Posizione: 6 + Base 6 (Base)

Padding //1 Posizione: 7 + Base 7 (Base) // NON UTILIZZARE

OSS (Riduzione dimensione Struct)

Nei primi esempi con la memorizzazione di variabili scalari, abbiamo notato che l'ordine influenza la dimensione dello struct poche' vanno rispettate le regole di allineamento. Una volta definita una struct il compilatore NON ricorda i campi per ottimizzare spazio, ma li memorizza nell'ordine in cui sono stati dichiarati, l'eventuale ottimizzazione e' affidata quindi al programmatore.

GDB

Preliminari

Definire comando go

In questo corso consigliamo di definire un comando custom chiamato go in gdb. Per definirlo occorre editare il file `.gdbinit` all'interno della home dell'utente. Ad esempio sulla VM BIAR, possiamo editarlo con:

```
$ geany /home/biar/.gdbinit
```

Inserire nel file il seguente contenuto:

```
define go
  start
  layout src
  layout regs
  focus cmd
end
```

Compilare programma con i simboli di debugging

Per facilitare la fase di debugging di un programma, dobbiamo compilare il binario utilizzando anche la flag `-g`. Ad esempio:

```
gcc -m32 e_main.c e.s -o e -g
```

Dove `e.s` contiene il nostro codice ASM e `e` sarà l'eseguibile generato.

Avviare il debugger

Per avviare la fase di debugging su un eseguibile `e`:

```
$ gdb ./e
```

Comandi

Avvio e terminazione:

```
(gdb) go // esegue uno script che lancia il debugging del programma e si ferma sulla prima istruzione del main
(gdb) run [<arg> ...] // per (ri)lanciare l'esecuzione con eventuali argomenti
(gdb) quit // per uscire da gdb (o CTRL-d)
```

Controllo esecuzione:

```
(gdb) cont // per riprendere l'esecuzione normalmente dopo un breakpoint
(gdb) step // prosegue l'esecuzione di una singola istruzione
(gdb) next // esegue istruzione in modo atomico: se è una chiamata a funzione, viene eseguita fino al suo return
(gdb) finish // esegue fino al termine della funzione corrente
```

Gestione breakpoint:

```
(gdb) break e.c:20 // inserisco un breakpoint alla linea 20 di e.c
(gdb) break e.s:20 // inserisco un breakpoint alla linea 20 di e.s
(gdb) info break // per mostrare breakpoint attivi
(gdb) clear e.s:20 // per eliminare breakpoint su una locazione di riferimento
(gdb) delete 1 // per eliminare il breakpoint contrassegnato come 1 da 'info break'
```

Ispezione dello stato:

```
(gdb) print $eax // stampa il contenuto di un registro
(gdb) print x // stampa il contenuto della variabile x nello stack frame corrente
(gdb) x/addr // stampa il contenuto della memoria all'indirizzo addr
(gdb) x/$eax + 4 // stampa il contenuto della memoria all'indirizzo dat dall'espressione (eax + 4)
(gdb) x/nfu addr // stampa il contenuto della memoria all'indirizzo addr secondo un formato.
```

Formato nfu (documentazione):

- `n`: quante volte si ripete un dato (repeat count), default=1, utile per stampare array
- `f`: tipo del dato ('x', 'd', 'u', 'o', 't', 'a', 'c', 'f', 's')
- `u`: dimensione del dato ('b': 1, 'h': 2, 'w': 4, 'g': 8)

Abbreviazioni dei comandi:

- print => p
- next => n
- cont => c
- step => s

Altri comandi:

```
(gdb) start // si posiziona all'inizio del main; non necessario quando si usa go
(gdb) file <nome_eseguibile> // per caricare un eseguibile
(gdb) break e.c:20 if x == 0 // breakpoint condizionale: si ferma solo se quando la riga 20 è eseguita x è uguale a zero
(gdb) frame n // selezione frame (utile per ispezionare lo stato)
(gdb) where // mostra la posizione corrente nell'esecuzione
(gdb) list // mostra il contenuto del file sorgente che contiene la funzione corrente
(gdb) backtrace // elenco frame attivi
```

21 MARZO

STACK FRAME E REGISTRO EBP

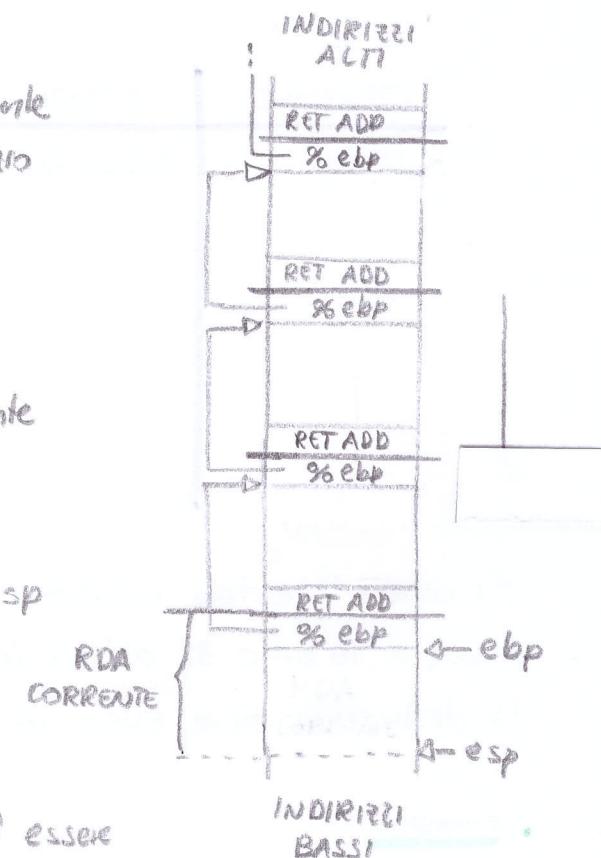
Il registro EBP e' un registro di tipo callee-save che viene utilizzato solitamente dal compilatore gcc per tenere traccia di tutti i record di attivazione (RDA) invocati da una funzione, tale procedura facilita il debugger.

Per utilizzare EBP a tale scopo e' sufficiente copiare il contenuto di %esp in %ebp all'inizio di ogni RDA, si ricorda che prima di tutto occorre salvare %esp nella stack dato che e' un registro callee-save.

In questo modo si crea una lista collegata avente come nodi i RDA.

Un altro aspetto interessante e' che nel RDA attualmente in esecuzione, una volta copiato %esp in %ebp, quest'ultimo punta all'inizio del RDA corrente. Ciò puo' essere usato come punto fisso per accedere agli elementi nella stack, nel mentre %esp puo' essere continuamente spostato.

Utilizzare %ebp per questo scopo NON e' obbligatorio, e risulta poco utile al programmatore se si adotta la tecnica PROLOGO - EPILOGO descritta precedentemente (%esp costante). Il compilatore infatti se impostato per ottimizzare il codice non usa %ebp per questo scopo, ma come un comune registro callee-save, e noi possiamo quindi fare lo stesso.



ISTRUZIONE CMOVcc (Assegnamento condizionato)

L'istruzione CMOVcc consente di effettuare degli assegnamenti solo se una determinata condizione è verificata. La condizione è descritta dal condition code cc, lo stesso visto per le istruzioni di salto Jcc.

ISTRUZIONE	EFFETTO	DESCRIZIONE
CMOVcc S,D	if(cc) D ← S	Se la condizione associata al suffisso cc è verificata, copia la sorgente S nella destinazione D. VINCOLI: S ≠ Immmediato D = Registro S,D = 16 o 32 bit

Si osservi che tale istruzione ha diversi vincoli: gli operandi devono essere a 16 bit o 32 bit, la sorgente non può essere un immmediato e la destinazione deve essere un registro.

esempio:

C: if(eax > ecx) ecx = eax;

ASH:

cmpl %ecx, %eax
cmovgl %eax, %ecx

OSS (suffisso)

Naturalmente come tutte le istruzioni ASH AT&T, la CMOVcc necessita del suffisso che specifica la dimensione degli operandi, in questo caso sono validi solo "w" e "l" (16 e 32 bit)

ISTRUZIONI DI SHIFT SHL, SHR, SAL, SAR

Le istruzioni di shift consentono di spostare a sinistra o a destra l'intero treno di bit di un operando.

La famiglia SHL, SHR effettua gli scorrimenti su operandi senza segno (shift logici), mentre SAL, SAR agiscono su operandi con segno (Shift aritmetici).

ISTRUZIONE	EFFETTO	DESCRIZIONE
SHL X,D SAL X,D	$D \leftarrow D \ll X$	Effettua shift logico o ARITMETICO a SINISTRA dell'operando D di X posizioni.
SHR X,D	$D \leftarrow D \gg X$	Effettua shift Logico a DESTRA dell'operando D di X posizioni.
SAR X,D	$D \leftarrow D \gg X$	Effettua shift ARITMETICO a DESTRA dell'operando D di X posizioni.

Tutte le precedenti istruzioni hanno i seguenti VINCOLI:

- D è un registro o memoria (8, 16, 32 bit)
- X può essere %cl, o un immediato
- Se $X=1$, l'operando X può essere omesso.

• SHIFT PER OPERAZIONI ARITMETICHE (SHIFT ARITMETICI)

Lo shift a DESTRA di X posizioni corrisponde ad una DIVISIONE per 2^X dell'operando D, mentre lo shift a SINISTRA di X posizioni corrisponde ad una moltiplicazione per 2^X dell'operando D.

$$D \leftarrow D \ll X \iff D \leftarrow D \cdot 2^X$$

$$D \leftarrow D \gg X \iff D \leftarrow D \cdot \frac{1}{2^X} \quad (\text{Valido solo con shift ARITMETICO})$$

Nel caso di shift verso sinistra, SHL e SAL sono equivalenti infatti sono associate al medesimo codice macchina. Al contrario gli shift a destra SHR, SAR sono diversi, infatti a causa della rappresentazione in complemento a due si effettua una corretta divisione solo tramite SAR. La SAR al momento dello shift a destra pone i nuovi bit uguali al bit più significativo, al contrario SHR pone sempre i nuovi bit a 0.

	(-64)	$\gg 3$	$(-8) = (-64)/2^3$
SAR	1 1 0 0 0 0 0 0	\rightarrow	1 1 1 1 1 0 0 0
SHR	1 1 0 0 0 0 0 0	\rightarrow	0 0 0 1 1 0 0 0

Oss (Uso shift)

Dato che le operazioni di moltiplicazione e divisione sono molto costose, il compilatore quando possibile usa gli shift.

ISTRUZIONE IDIV

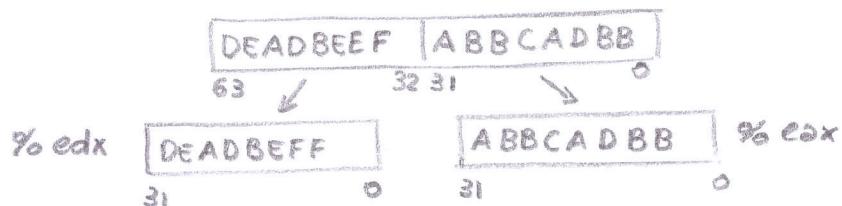
L'istruzione IDIV consente di calcolare una divisione INTERA ed il suo resto contemporaneamente, tale istruzione ha un solo operando che specifica il divisore, il dividendo invece è sempre dato dalla concatenazione di %edx ed %eax. (/ → Divisore ; % → Resto)

ISTRUZIONE	EFFETTO	DESCRIZIONE
IDIV V	$A \leftarrow DA/V$ $D \leftarrow DA \% V$	Calcola simultaneamente il quoziente ed il resto della divisione del dividendo ottenuto concatenando D con A, dove D ha i bit più significativi, con il divisore V che può essere un operando registro o memoria.

L'istruzione IDIV ha le seguenti varianti in relazione al suffisso:

ISTRUZIONE	EFFETTO
idivb V ₁	$\%al \leftarrow \%ax / V_1$ $\%ah \leftarrow \%ax \% V_1$
idivw V ₂	$\%ax \leftarrow \%dx \%ax / V_2$ $\%dx \leftarrow \%dx \%ax \% V_2$
idivl V ₄	$\%eax \leftarrow \%edx \%eax / V_4$ $\%edx \leftarrow \%edx \%eax \% V_4$

Dato che IDIV usa due registri a 32 bit ci consente di avere un dividendo a 64 bit, occorre tuttavia rappresentare il dato correttamente. Se il dato memorizzato e' a 64 bit e' sufficiente mettere i bit più significativi in %edx e quelli meno in %eax:



Se invece il dato e' a 32 bit occorre promoverlo a 64 bit su due registri facendo attenzione ad estenderlo correttamente. (dato signed) Poiché l'estensione e' su due registri separati non possono usare MOVS, si utilizzano invece lo shift aritmetico (SAR) nel seguente modo:

$$X = 0x DEADBEEF$$

%edx DEADBEEF

%eax DEADBEEF

$\downarrow >> 31$ (SAR)

%edx

FFFF FFFF

DEADBEEF

%eax

FFFF FFFF DEADBEEF

%edx %eax

Rende coerente il dato 32 bit su 64 bit

```
// esempio ASH X/V
movl X, %eax
movl %eax, %edx
sarl $31, %edx
idivl V
```


OTTIMIZZAZIONI DELLE PRESTAZIONI DEL SOFTWARE

Durante la loro esecuzione, i programmi usano la CPU e la memoria del calcolatore, ovvero le RISORSE del sistema, quest'ultime sono contese da diversi processi.

Quando si ha l'obiettivo di ridurre il consumo della risorsa CPU (risorsa di calcolo) si parla di OTTIMIZZAZIONE DELLA RISORSA TEMPO.

Se invece si vuole ridurre il consumo della risorsa memoria si effettua un'OTTIMIZZAZIONE DI SPAZIO.

• PROPRIETA' DEL SOFTWARE

Le proprietà più comuni di un software sono le seguenti:

- CORRETTEZZA
- LEGGIBILITÀ
- ROBUSTEZZA
- MODULARITÀ
- RIUTILIZZABILITÀ

- PRESTANZE: Buone prestazioni, alta velocità di esecuzione

Soltanmente all'aumentare di una certa proprietà c'è una riduzione delle altre. Ovviamente la più importante che dovrà essere sempre presente è la CORRETTEZZA.

• TIPI DI OTTIMIZZAZIONE

Per ottimizzare un software si può ottimizzare l'ALGORITMO oppure il PROGRAMMA che lo implementa.

Quando possibile, conviene ottimizzare l'algoritmo infatti se un algoritmo ha costo $O(n^2)$ raramente si riesce ad implementare un programma ottimizzato con costo minore, ad esempio $O(n)$.

Ottimizzando il programma riesco solo a cambiare le costanti moltiplicative del costo asintotico dell'algoritmo e NON la complessità.

L'ottimizzazione puo' essere effettuata da:

- Programmatore
- Compilatore

Il compilatore puo' applicare ottimizzazioni sul programma, ma non sull'algoritmo. Il programmatore dovrà quindi ottimizzare prima di tutto l'algoritmo e successivamente "consentire" al compilatore di ottimizzare il programma.

Il compilatore gcc consente diversi livelli di ottimizzazione impostabili tramite la flag -O, per vedere le singole opzioni da terminale:

>> man gcc

Ad esempio la rimozione dell'uso di %ebp nelle chiamate a funzione si ha solitamente con l'opzione -O1, oppure direttamente tramite -fomit-frame-pointer

TECNICA DI OTTIMIZZAZIONE CODICE (RIDUZIONE WORK)

Una tecnica di ottimizzazione del codice consiste nel ridurre il numero di istruzioni eseguite, ovvero RIDURRE IL WORK

Vediamo alcune tecniche di questo tipo che vengono applicate dal compilatore:

• CONSTANT FOLDING

La tecnica di CONSTANT FOLDING (Ripiegamento delle costanti) consiste nel sostituire espressioni con operandi costanti con il risultato dell'espressione, in questo modo le espressioni NON sono calcolate durante l'esecuzione del programma.

- esempio

$$\text{int } x = 2 + 3 \longrightarrow \text{int } x = 5$$

$$\text{double } x = 3 * \pi \longrightarrow \text{double } x = 3 * 3.14 = \text{double } x = 9.42$$

- CONSTANT PROPAGATION

(La tecnica di) **CONSTANT PROPAGATION** (propagazione delle costanti)

Si applica quando ad una variabile è assegnato un valore costante, e consiste nel sostituire la variabile con tale costante e successivamente applicare il **constant folding**.

- Esempio

const int x = 7

int y = x * 3 → int y = 7 * 3 → int y = 21

- DEAD CODE ELIMINATION

Tale tecnica consiste nell'eliminare porzioni di codice che non verranno mai eseguiti; si ha un caso di questo tipo quando la condizione di un blocco if là si conosce a tempo di compilazione.

- Esempio

int x = 7

if(x > 1) return 10; → int x = 7
return -1;

- COMMON SUBEXPRESSION ELIMINATION

Espressioni complesse che contengono al loro interno sottosexpressioni ripetute possono essere semplificate calcolando separatamente le sottosexpressioni comuni e risuonando il valore calcolato. Tale tecnica viene chiamata **COMMON SUBEXPRESSION ELIMINATION**.

int a, b

...

int x = (a - b) * (a - b)

→

int a, b

...

int temp = (a - b)

int x = temp * temp

Le precedenti tecniche sono effettuate automaticamente da gcc con la specifica -O1. Vediamo altre tecniche che non sempre possono essere applicate automaticamente.

• LOOP-INVARIANT CODE MOTION

Il LOOP-INVARIANT CODE MOTION (o hoisting) consiste nell'identificare porzioni di codice contenute in un ciclo che ad ogni iterazione eseguono lo stesso calcolo invariante, e di spostarle all'esterno, prima del ciclo.

Tale tecnica è applicata automaticamente solo se gcc conosce la funzione chiamata oppure è un'espressione invariante.

- Esempio (Applicazione automatica)

In questo caso gcc conosce la funzione sqrt(), quindi può applicare automaticamente la precedente tecnica

```
while (i < n){  
    double d = sqrt(x);  
    s = s + d;  
    i++;  
} → double d = sqrt(x)  
      while (i < n)  
          s = s + d;  
          i++;
```

- Esempio (Applicazione NON automatica)

Nel seguente caso invece gcc non conosce strlen() e quindi per evitare di modificare la semantica del programma, non effettua l'ottimizzazione.

```
char* s = "hello"  
int i  
for (i=0; i < strlen(s); i++)  
    putchar(s[i])
```

MANUALMENTE →

```
char* s = "hello"  
int i, len = strlen(s)  
for (i=0; i < len; i++)  
    putchar(s[i])
```

Tale ottimizzazione, in questi casi, è applicabile solo dal programmatore.

• FUNCTION INLINING

Quando una funzione viene chiamata ripetutamente, ad esempio in un ciclo, ed il suo corpo contiene poche istruzioni, può essere vantaggioso applicare il FUNCTION INLINING, ovvero riempiazzare una chiamata a funzione con il corpo della funzione chiamata. Tale tecnica può essere applicata dal compilatore solo se la funzione chiamata è contenuta nello stesso file sorgente.

```
int sum(int x, int y)
```

```
    return x+y;
```

```
int main()
```

```
...
```

```
int z = sum(x, y)
```



```
int main()
```

```
...
```

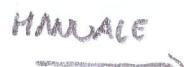
```
int z = x+y
```

Nel caso in cui la definizione della funzione è esterna al file, poiché usata da più moduli, gcc non può applicare tale tecnica. Per consentire tale ottimizzazione si può copiare la funzione chiamata nel file sorgente, cambiandone il nome e rendendola statica in modo da non essere visibile dagli altri moduli che useranno quella iniziale.

```
// sum.c
```

```
int sum(int x, int y)
```

```
    return x+y;
```



```
// main.c
```

```
int main()
```

```
...
```

```
int z = sum(x, y)
```

```
// sum.c (INVARIATO)
```

```
int sum(int x, int y)
```

```
    return x+y
```

```
// main.c
```

```
int static sum_2(int x, int y)
```

```
    return x+y
```

```
int main()
```

```
...
```

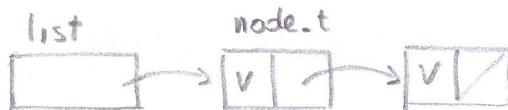
```
int z = sum_2(x, y)
```

A questo punto, dopo la modifica, gcc può applicare la function inlining

• TECNICA AUGMENTATION (aumento)

Tale tecnica consiste nel ridurre il numero di istruzioni a discapito della memoria, per descrivere tale tecnica consideriamo la seguente struct:

```
typedef struct node_t {
    int v
    node_t* next
} node_t
struct list {
    node_t* head
}
```



Venfies eselleem

Con tale implementazione si ha accesso con costo costante all'elemento in testa, e costo lineare per quelli in coda.
Inoltre anche determinare la lunghezza della lista ha costo lineare.

Costo accesso head = O(1)
Costo accesso tail = O(n)
Costo size = O(n)

Per migliorare le prestazioni possiamo aggiungere due nuovi campi alla struct list, uno contenente il puntatore alla coda e un altro contenente la dimensione, in questo modo otteniamo costi costanti per le precedenti operazioni:

```
struct list {
    node_t* head
    node_t* tail
    int size
}
```

Costo accesso head = O(1)
Costo accesso tail = O(1)
Costo size = O(1)

Si osservi che in questo modo abbiamo migliorato notevolmente i costi delle operazioni di base, ma come conseguenza si ha un maggiore uso dello spazio di memoria ed inoltre occorre gestire correttamente il nuovo campo size, ovvero incrementarlo ad ogni inserimento e decrementarlo ad ogni rimozione.

Un'altra operazione che è possibile migliorare è la scansione della lista, con la precedente implementazione l'accesso all'elemento precedente rispetto al nodo attuale ha costo lineare poiché occorre scorrere nuovamente la lista. Aggiungendo nella struct node-t un nuovo campo contenente il puntatore al nodo precedente, si ottiene un costo costante.

```
typedef struct node-t {
    int v
    Node-t* next
    Node-t* prev
} Node-t
```

Costo accesso tail = O(1)
Costo accesso head = O(1)
Costo size = O(1)
Costo iteratore = O(1)

• LOOP UNROLLING (strobolamento del ciclo)

Questa tecnica ha lo scopo di ridurre il numero di cicli eseguiti, per fare ciò si esegue più di un iterazione in un singolo ciclo. Si consideri il seguente ciclo che somma gli elementi di un array:

```
int s=0
int i;
for(i=0; i<n; i++)
    s += v[i];
```

In questo caso applicando il loop unrolling si ottiene:

```
...
for(i=0; i+3<n; i+=4)
    s += v[i]+v[i+1]+v[i+2]+v[i+3]
for(; i<n; i++) // somma i restanti elementi.
    s += v[i]
```

In questo caso si dice loop unrolling di RAGIONE 4, poiché ad ogni ciclo vengono eseguite quattro iterazioni.

OSS (for finale)

Si osservi che l'utilizzo di soltanto il primo for NON rende il programma ottimizzato corretto semanticamente, poiché produrrebbe un risultato corretto solo se V ha dimensione multiplo di quattro. Il secondo ciclo serve appunto per considerare gli elementi rimanenti, che saranno al più tre.

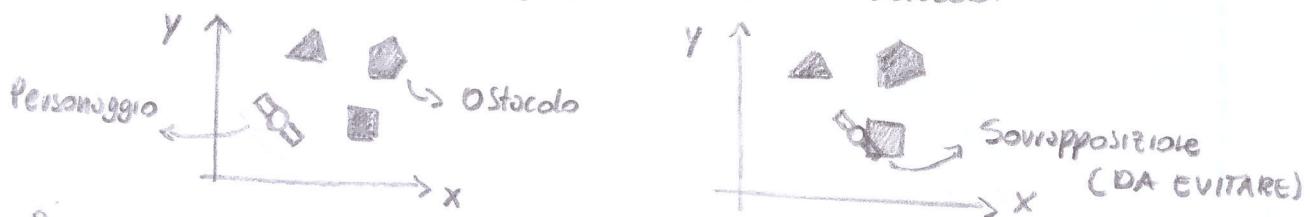
Tale ottimizzazione è eseguita dal compilatore solamente se si conosce, a tempo di compilazione, la dimensione n .

• IDENTITA' ALGEBRICHE (Algebraic Identity)

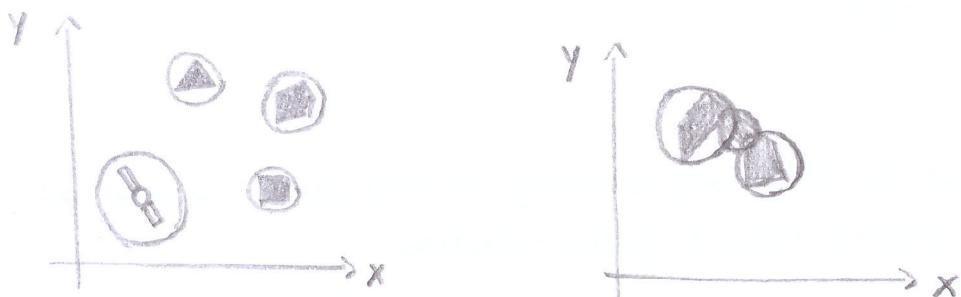
Molto spesso le identità algebriche consentono di ridurre o semplificare determinati calcoli, vediamo un esempio in un videogioco 2D.

- Esempio (videogioco 2D)

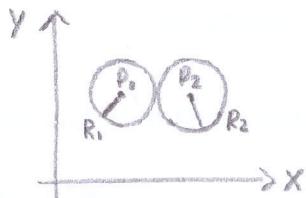
Si consideri un videogioco 2D in cui un personaggio si muove in un piano contenente ostacoli, il nostro obiettivo è evitare che un personaggio si sovrapponga graficamente ad un ostacolo.



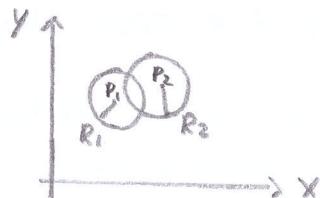
Per evitare la sovrapposizione, considerando le reali forme degli oggetti o del personaggio, si impiegano troppe operazioni, per semplificare tali calcoli si approssimano tutti gli elementi a semplici circonference che circondano gli oggetti. Nel caso di oggetti complessi li si considera come formati da più cerchi.



La scelta di approssimare con delle circonference e' dovuta al fatto che in questo modo si avra' una collisione se e solo se la distanza tra i centri di due circonference e' minore o uguale alla somma dei raggi.



$$\text{dist}(P_1, P_2) = R_1 + R_2$$



$$\text{dist}(P_1, P_2) < R_1 + R_2$$

Dato che ci muoviamo su un piano x, y :

$$P_1 = (x_1, y_1) ; P_2 = (x_2, y_2) ; d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Collisione se e solo se $d < R_1 + R_2$

Un implementazione in C e' la seguente:

```
typedef struct circle {
    double x, y, r;
} circle_t;
```

L'operazione sqrt, ovvero il calcolo della radice quadrata di un numero e' un istruzione onerosa, sfruttando la seguente semplice identita' algebrica la si puo' rimuovere.

$$\begin{aligned} d < R_1 + R_2 &\Leftrightarrow d^2 < (R_1 + R_2)^2 \\ &\Leftrightarrow (x_1 - x_2)^2 + (y_1 - y_2)^2 < (R_1 + R_2)^2 \end{aligned}$$

```
int collision(circle_t* c1, circle_t* c2) {
    double d = sqrt(
        ((c1->x - c2->x) * (c1->x - c2->x)) +
        ((c1->y - c2->y) * (c1->y - c2->y)));
    return d < (c1->r + c2->r);
    // ritorna true se colloidano
```

```
// Sfrutta identita' algebrica
int collision(circle_t* c1, circle_t* c2) {
    double d2 =
        ((c1->x - c2->x) * (c1->x - c2->x)) +
        ((c1->y - c2->y) * (c1->y - c2->y));
    return d2 < ((c1->r + c2->r) *
                  (c1->r + c2->r));
```

• TECNICA DELLA CORTOCIRCUITAZIONE

In tale tecnica si sfrutta la circuitazione delle condizioni bodeate in C:

A && B → Se A e' FALSE allora NON valuta B

A || B → Se A e' TRUE allora NON valuta B

Un esempio in cui si puo' sfruttare tale tecnica e' nel dover riconoscere se un char c e' un carattere di tabulazione ('\t', ' ', '\n') o meno. Si puo' ottimizzare tale verifica mettendo diversi OR in cascata in ordine dal piu' probabile (piu' frequente), a quel punto meno probabile, in questo modo se char c e' un comune spazio ' ', si esegue un solo confronto, ignorando gli altri grazie alla cortocircuitazione del c.

```
int is-blank(char c){  
    return c == '\t' || c == ' '  
        || c == '\n'; }  
// NON OTTIMIZZATO
```

```
int is-blank(char c) {  
    return c == ' ' || c == '\n'  
        || c == '\t';}  
// OTTIMIZZATO
```

• COMPILE TIME INITIALIZATION (Inizializzazione a tempo di compilazione)

Tale tecnica consiste nell'evitare di eseguire calcoli per determinate una o piu' caratteristiche di un dato, per farlo si usano le TABELLE DI LOOKUP che essenzialmente sono matrici o array che contengono i risultati di tutti i possibili calcoli che il programma potrebbe fare in esecuzione.

Le tabelle di lookup si utilizzano quando accedervi e' meno oneroso di eseguire il calcolo a runtime.

- Esempio (Consonanti)

Un esempio banale di uso di tabella di lookup e' per determinare se un carattere char c e' una consonante o meno.

La funzione senza tabella di lookup e' la seguente e semplicemente confronta char c con le vocali:

```
// No look-up table  
int is_consonante(char c){  
    return (c >= 'b' && c <= 'z') || (c >= 'B' && c <= 'Z')  
        && (c != 'e' && c != i && c != o && c != u)  
        && (c != 'E' && c != I && c != O && c != U);
```

Per evitare tutti i precedenti confronti usiamo una tabella di lookup costituita da un array di 256 interi $\in \{0, 1\}$ che in corrispondenza dell'indice pari al codice ASCII di c restituisce 0 se c e' una vocale e 1 se e' una consonante:

```
int [256] consonanti = {0, ..., 0, 1, 1, ..., 0, ... 0, 1, ...}  
                           (B) (C)   (E)   (A) (b)
```

Di conseguenza, avendo a disposizione tale tabella (array), la funzione diventa immediata e priva di confronti.

```
// look-up table  
int is_consonante(char c)  
    . . . return consonanti[c];
```

- Esempio (Numeri Primi) (L11-190601)

Un esempio molto piu' utile nella pratica e' quello di utilizzare una tabella di lookup (array) per memorizzare un determinato insieme di numeri primi. Si ricorda infatti che stabilire se un numero e' primo o meno e' un operazione onerosa.

Nell'esempio si hanno due implementazioni, una con la tabella di lookup e l'altra senza, tramite il seguente comando si misura il tempo di esecuzione di un programma:

```
>> time ./example
```

TECNICA DI OTTIMIZZAZIONE CODICE: (RIDUZIONE COSTO ISTRUZIONI)

Un secondo approccio per velocizzare un programma consiste nel rimpiazzare istruzioni con alternative più veloci.

Vediamo due esempi.

• REGISTER ALLOCATION

Tale ottimizzazione consiste nel memorizzare le variabili del programma direttamente nei registri general purpose evitando di memorizzare le variabili locali nello stack poiché l'accesso a memoria è molto costosa.

In C esiste la parola chiave REGISTER che richiede al compilatore di memorizzare la variabile associata in un registro, si osservi che con tale direttiva non si ha nessuna garanzia.

```
{ register int x; // Il compilatore PROVA ad  
// utilizzare un registro.
```

Tale direttiva NON è molto utilizzata dato che i moderni compilatori cercano sempre di usare solo registri.

• STRENGTH REDUCTION

La seguente tecnica consiste nel sostituire un'istruzione onerosa con una o più istruzioni meno onerose che eseguono la medesima operazione. Ad esempio, il prodotto e la divisione possono essere sostituiti da somme, sottrazioni, shift o LEA

es:

$$m * 2^n \longleftrightarrow m \ll n \text{ (SHL)}$$

$$m / 2^n \longleftrightarrow m \gg n \text{ (SAR)}$$

$$x * 15 \longleftrightarrow x * 16 - x = (x \ll 4) - x$$

$$x * 17 \longleftrightarrow x * 16 + x = (x \ll 4) + x$$

TECNICHE PER OTTIMIZZARE LO SPAZIO

Come visto precedentemente la memoria deve rispettare un allineamento perciò un oggetto di dimensione s deve avere un indirizzo x che è multiplo di s ($x \bmod s = 0$). L'allineamento viene effettuato ovunque vengano memorizzati oggetti nello spazio di memoria di un processo:

- Data (Variabili Globali)
- Stack Frame
- Heap
- Codice (Indirizzo funzioni)
- Strutture

Una tecnica per ridurre lo spazio occupato dalle strutture elencate, vista in precedenza (ASH), consiste nell'ordinare i campi in modo decrescente rispetto alla loro dimensione, in questo modo si riducono i byte di padding.

es:

```
struct s {  
    int b  
    int a  
    short c  
    char d }
```



N.B. (Specifico gcc -Os)

Si noti che gcc con l'estensione per la riduzione dello spazio -Os non può spostare i campi di una struct in modo da diminuire i byte di padding poiché il programmatore potrebbe aver fatto affidamento sulla loro posizione in memoria.

Ridurre i byte di padding in una struct è quindi un'ottimizzazione che può eseguire solo il programmatore.

INDIVIDUAZIONE HOT SPOT

Nell'ottimizzare un programma occorre innanzitutto conoscere i suoi HOT SPOT (o colpi di bottiglia) ovvero le funzioni che influiscono maggiormente sul tempo di esecuzione del programma, in questo modo possiamo effettuare un'ottimizzazione mirata. Come in molti altri campi, vale la legge del 20/80, ovvero molto spesso c'è una piccola parte del programma (20%) che viene eseguito per la maggior parte del tempo (80%).

• PERFORMER PROFILER

Per individuare la percentuale di tempo per cui è eseguita una funzione e quindi determinare gli hotspot, si utilizzano programmi di ausilio detti PERFORMER PROFILER, noi utilizzeremo GProf appartenente allo suite GNU. GProf si utilizza nel seguente modo:

- 1- Si compila, tramite gcc, con lo flag -pg. In questa fase il compilatore aggiunge il codice per effettuare i successivi calcoli.

```
>> gcc -O e1 e1.c -pg
```

- 2- Si lancia l'eseguibile generato, tale programma oltre a contenere il nostro codice, contiene del codice aggiuntivo per produrre un binario interpretabile da GProf

```
>> ./e1
```

- 3- Come ultimo passo si lancia gprof con argomento il nome dell'eseguibile generato al primo passo.

```
>> gprof e1
```

Di default GProf stampa il report a video, alternativamente possiamo "streammare" il report su file con l'aggiunta di "> file"

```
>> gprof e1 > report.txt
```

```

1  Flat profile:
2  Each sample counts as 0.01 seconds.
3      % cumulative   self           self      total
4      time   seconds   seconds   calls  ms/call  ms/call  name
5      50.36     3.52     3.52    100000     0.04     0.04  B
6      49.86     7.00     3.48    100000     0.03     0.03  A
7      0.07     7.00     0.01         1      5.02     5.02 init
8
9      %           the percentage of the total running time of the
10     time        program used by this function.
11
12    cumulative a running sum of the number of seconds accounted
13    seconds   for by this function and those listed above it.
14
15    self       the number of seconds accounted for by this
16    seconds   function alone. This is the major sort for this
17    listing.
18
19    calls      the number of times this function was invoked, if
20    this function is profiled, else blank.
21
22    self       the average number of milliseconds spent in this
23    ms/call    function per call, if this function is profiled,
24    else blank.
25
26    total      the average number of milliseconds spent in this
27    ms/call    function and its descendants per call, if this
28    function is profiled, else blank.
29
30    name       the name of the function. This is the minor sort
31          for this listing. The index shows the location of
32          the function in the gprof listing. If the index is
33          in parenthesis it shows where it would appear in
34          the gprof listing if it were to be printed.
35 FF
36 Copyright (C) 2012-2018 Free Software Foundation, Inc.
37 Copying and distribution of this file, with or without modification,
38 are permitted in any medium without royalty provided the copyright
39 notice and this notice are preserved.
40 FF Call graph (explanation follows)
41
42 granularity: each sample hit covers 2 byte(s) for 0.14% of 7.00 seconds
43
44 index % time   self  children   called   name
45                               <spontaneous>
46 [1]   100.0   0.00    7.00           main [1]
47           3.52   0.00  100000/100000     B [2]
48           3.48   0.00  100000/100000     A [3]
49           0.01   0.00         1/1       init [4]
50 -----
51           3.52   0.00  100000/100000     main [1]
52 [2]   50.2    3.52   0.00  100000     B [2]
53 -----
54           3.48   0.00  100000/100000     main [1]
55 [3]   49.7    3.48   0.00  100000     A [3]
56 -----
57           0.01   0.00         1/1       main [1]
58 [4]    0.1     0.01   0.00         1       init [4]
59 -----
60

```

61 This table describes the call tree of the program, and was sorted by
62 the total amount of time spent in each function and its children.
63

64 Each entry in this table consists of several lines. The line with the
65 index number at the left hand margin lists the current function.
66 The lines above it list the functions that called this function,
67 and the lines below it list the functions this one called.
68 This line lists:

69 index A unique number given to each element of the table.
70 Index numbers are sorted numerically.
71 The index number is printed next to every function name so
72 it is easier to look up where the function is in the table.
73

74 % time This is the percentage of the 'total' time that was spent
75 in this function and its children. Note that due to
76 different viewpoints, functions excluded by options, etc,
77 these numbers will NOT add up to 100%.

78

79 self This is the total amount of time spent in this function.
80

81 children This is the total amount of time propagated into this
82 function by its children.

83

84 called This is the number of times the function was called.
85 If the function called itself recursively, the number
86 only includes non-recursive calls, and is followed by
87 a '+' and the number of recursive calls.

88

89 name The name of the current function. The index number is
90 printed after it. If the function is a member of a
91 cycle, the cycle number is printed between the
92 function's name and the index number.

93

94

95 For the function's parents, the fields have the following meanings:

96

97 self This is the amount of time that was propagated directly
98 from the function into this parent.

99

100 children This is the amount of time that was propagated from
101 the function's children into this parent.

102

103 called This is the number of times this parent called the
104 function '/' the total number of times the function
105 was called. Recursive calls to the function are not
106 included in the number after the '/'.

107

108 name This is the name of the parent. The parent's index
109 number is printed after it. If the parent is a
110 member of a cycle, the cycle number is printed between
111 the name and the index number.

112

113 If the parents of the function cannot be determined, the word
114 '<spontaneous>' is printed in the 'name' field, and all the other
115 fields are blank.

116

117 For the function's children, the fields have the following meanings:

118

119 self This is the amount of time that was propagated directly
120 from the child into the function.

121 children This is the amount of time that was propagated from the
122 child's children to the function.
123
124 called This is the number of times the function called
125 this child `/' the total number of times the child
126 was called. Recursive calls by the child are not
127 listed in the number after the `/'.
128
129 name This is the name of the child. The child's index
130 number is printed after it. If the child is a
131 member of a cycle, the cycle number is printed
132 between the name and the index number.
133
134 If there are any cycles (circles) in the call graph, there is an
135 entry for the cycle-as-a-whole. This entry shows who called the
136 cycle (as parents) and the members of the cycle (as children.)
137 The `+' recursive calls entry shows the number of function calls that
138 were internal to the cycle, and the calls entry for each member shows,
139 for that member, how many times it was called from other members of
140 the cycle.
141
FF
142 Copyright (C) 2012-2018 Free Software Foundation, Inc.
143
144 Copying and distribution of this file, with or without modification,
145 are permitted in any medium without royalty provided the copyright
146 notice and this notice are preserved.
147
FF
148 Index by function name
149
150 [3] A [2] B [4] init
151
152

SPEED-UP

Un altro aspetto importante e' determinare di quanto aumenta la velocita' di un programma in seguito all'ottimizzazione di una funzione A che e' eseguita per una percentuale α del tempo totale.

DEF (Speed-Up)

Sia T il tempo di esecuzione di un programma e sia T' , il tempo di esecuzione dopo aver applicato un'ottimizzazione.

Si definisce SPEED-UP il rapporto tra T e T' , tale grandezza caratterizza il miglioramento prestazionale dovuto all'ottimizzazione.

$$\boxed{\text{Speed-up} = \frac{T}{T'}}$$

T = Tempo esecuzione P
 T' = Tempo esecuzione P OTTIMIZZATO

Per mantenere consistente la precedente definizione, cioe' per indicare un vero miglioramento prestazionale e' necessario che T' sia minore di T , quindi $s > 1$:

$$\text{Speed-up} = \frac{T}{T'} > 1$$

Soltamente un speed-up s si indica con Sx

es

Si consideri un programma P che impiega un tempo pari a 4 secondi ($T=4$), una volta ottimizzato (P') impiega un tempo $T' = 2$ secondi.
Si ha quindi:

$$\text{Speed-up} = \frac{T}{T'} = \frac{4}{2} = 2 \implies \text{Speed-up } 2x$$

LEGGE DI AHDAL

Supponiamo di dividere un programma in due parti A e B

P	A	B
---	---	---

Sia T il tempo totale speso dal programma, sia α la percentuale di T in cui viene eseguita A, ovvero:

$$T_A = \alpha T \quad \Rightarrow \quad T_B = (1 - \alpha) T$$

$\left| \begin{array}{l} \alpha = \text{Percentuale} \\ \in [0, 1] \end{array} \right.$

Supponiamo di ottimizzare A in modo che sia k volte più veloce

$$(A \rightarrow A') \quad \text{SpeedUp}(A) = \frac{T_A}{T_{A'}} = k$$

Lo speedUp del programma complessivo sarà dunque il seguente:

$$\text{SpeedUp}(P) = \frac{T}{T'} = \frac{T_A + T_B}{T_{A'} + T_B} = \frac{T_A + T_B}{\frac{T_A}{k} + T_B}$$

$$\left| \frac{T_A}{T_{A'}} = k \Leftrightarrow T_{A'} = \frac{T_A}{k} \right.$$

$$= \frac{\alpha T + (1 - \alpha) T}{\frac{\alpha T}{k} + (1 - \alpha) T} = \frac{1}{\frac{\alpha}{k} + 1 - \alpha}$$

Quest'ultimo risultato è detto LEGGE DI AHDAL e rappresenta quindi lo speed-up del programma in conseguenza ad uno speed-up $k \times$ di una funzione che costituisce una percentuale α del tempo complessivo.

$$\boxed{\text{Speed-up} = \frac{T}{T'} = \frac{1}{\frac{\alpha}{k} + 1 - \alpha}}$$

$\left| \begin{array}{l} \alpha = \% T \text{ della funzione opt} \\ k = \text{speed-up funzione opt} \end{array} \right.$

OSS

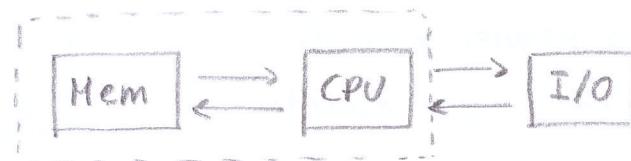
Lo speed-up massimo ottenibile, una volta fissato α , è dato da:

$$\lim_{k \rightarrow +\infty} \frac{1}{\frac{\alpha}{k} + 1 - \alpha} = \frac{1}{1 - \alpha}$$

Si noti quindi che più piccolo è α e meno ha senso ottimizzare la funzione associata

MODALITA' DI ESECUZIONE DELLA CPU (Mode)

Finora abbiamo visto programmi ASH che agiscono solo sulla CPU e sulla memoria del calcolatore, ma non abbiamo ancora usato istruzioni per la gestione delle risorse di input-output (I/O).



Accedere alle risorse di I/O, come ad esempio il disco rigido, è un'operazione "rischiosa" che se non eseguita correttamente può portare al danneggiamento dell'intero sistema.

Per questo motivo un processo utente (programma) non deve poter accedere direttamente a tali risorse e per impedirlo la CPU ha due MODALITA' DI ESECUZIONI:

- User Mode: La CPU esegue solo le istruzioni NON "pericolose"
- Kernel Mode: La CPU esegue QUALSIASI istruzione.

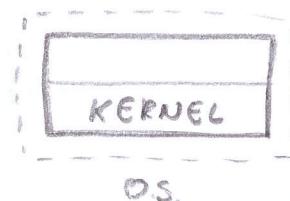
Naturalmente le istruzioni delle due modelli appartengono allo stesso ISA (Instruction Set Architecture).

La CPU ha una flag interna che abilita una delle due modelli, finora quindi i nostri programmi (processi) sono stati sempre eseguiti in User Mode.

• SISTEMA OPERATIVO

Dato che un processo non puo' accedere direttamente all'hardware della macchina vi si pone un intermediario ovvero il SISTEMA OPERATIVO (O.S.).

Un sistema operativo e' costituito essenzialmente da due parti, una detta KERNEL eseguita appunto in Kernel mode, ed un'altra eseguita in User Mode. Un esempio di Kernel e' LINUX.



L'O.S. offre quindi un ambiente protetto per i processi, garantisce infatti:

- PROTEZIONE: La macchina NON viene danneggiata da accessi NON corretti alle risorse I/O.

- ISOLAMENTO: Il singolo processo NON avverte la presenza di altri programmi in esecuzione.

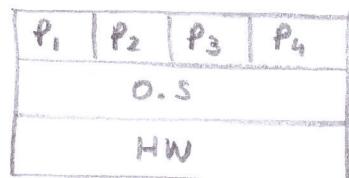
Ad esempio se vengono eseguiti tre processi in contemporanea, il primo processo P_1 non si accorge dell'interruzione dovuta a P_2 e P_3 .



Inoltre un processo NON puo' accedere da norma alla memoria di un altro processo.

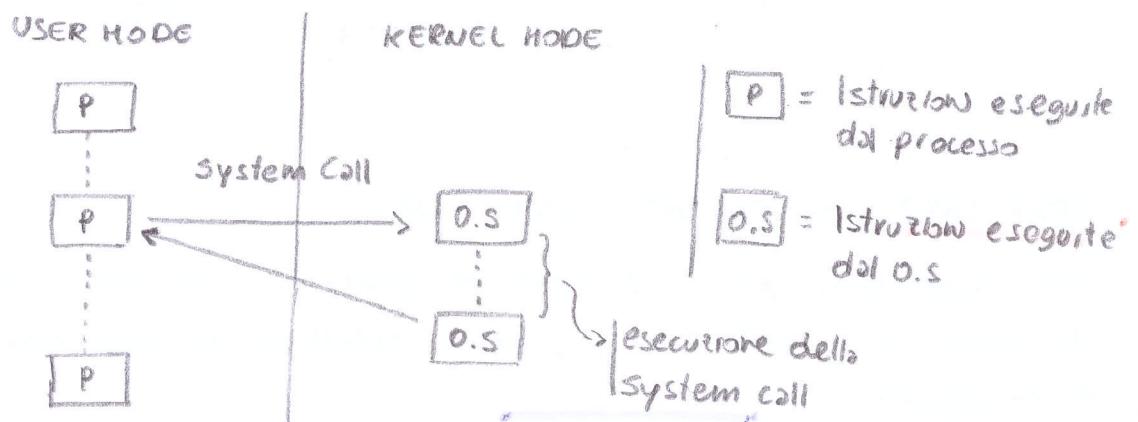
L'O.S. inoltre mette a disposizione diversi servizi per permettere ai processi di accedere in modo sicuro al I/O.

Un processo puo' utilizzare tali servizi messi a disposizione, tramite le SYSTEM CALL



SYSTEM CALL

Una chiamata a sistema, ovvero SYSTEM CALL, è simile ad una normale chiamata a funzione eccetto per il fatto che le istruzioni che vengono eseguite successivamente sono operate dal O.S. in Kernel Mode.

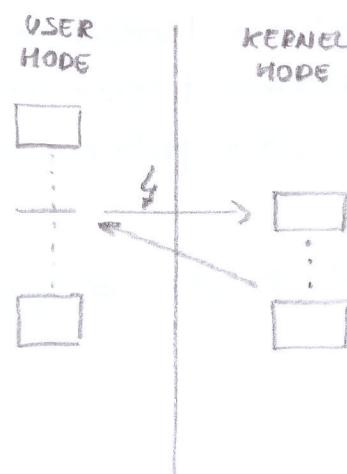


Le system call si basano sull'INTERRUZIONE del flusso di esecuzione del processo in modo da lasciare il controllo al O.S.
Vi sono due tipi di interruzione:

- INTERRUPT (HW) (ASINCRONO)
- TRAP (SW) (SINCRONO)

Interrupt

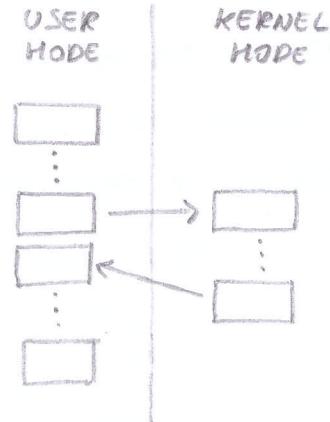
Tale interruzione è generata dal hardware (HW) in risposta a determinati eventi come ad esempio il click del mouse, completamento di un accesso I/O, la pressione di un tasto o il termine di un timer. L'obiettivo di tali interrupt HW è quello di evitare alla CPU di dover attendere, senza poter fare altro, uno dei precedenti eventi. L'interrupt HW è ASINCRONO rispetto al processo poiché non è determinato da quest'ultimo, ma appunto dal HW, che di conseguenza lascia il controllo al O.S.



- Trap

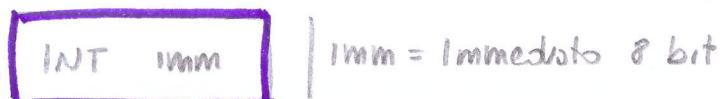
Questo tipo di interruzione è generata dal processo stesso (sw), volontariamente per passare il controllo al o.s.

Tale interruzione e' SINCRONA poiche' allineata (sincronizzata) con il processo.



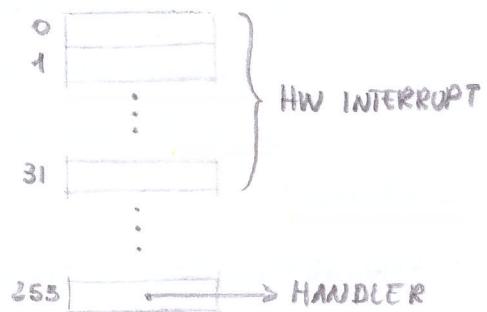
TRAP 1AB2 (SW INTERRUPT)(SYSTEM - CALL)

Nell'architettura IA32 l'istruzione per generare il passaggio volontario del flusso di controllo al O.S e quindi effettuare un'interruzione e' la seguente.



Tale istruzione ha solo un operando di tipo immediato a 8bit che rappresenta l'indice del INTERRUPT VECTOR.

L'Interrupt Vector contiene gli indirizzi degli HANDLER (gestori) dei diversi tipi di interrupt che possono essere generati.

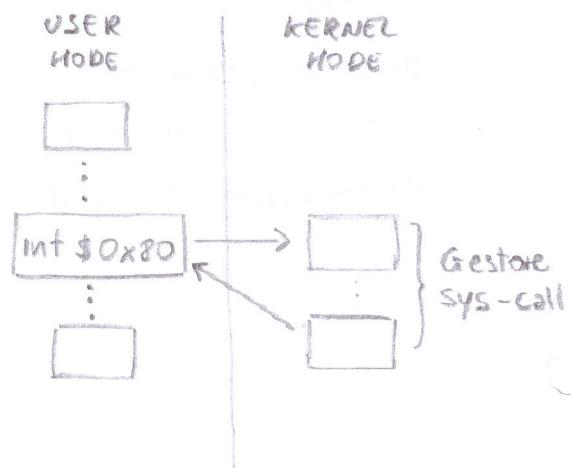


le prime 32 posizioni sono dedicate agli interrupt o tipi HW e quindi non ha senso invocarli tramite INT.

Nelle successive posizioni vi sono gli indirizzi degli handler per la gestione dei SW interrupt, tra cui quelli per le SYSTEM CALL.

Nei O.S che adottano lo stesso standard di GNU/Linux l'indice corrispondente alle system call e' quello in posizione 128 (0x80)

```
int $0x80 # System Call
```



• SELEZIONE SYSTEM CALL

Tramite INT \$0x80 possiamo invocare una system call, ma occorre specificare quale, per farlo utilizziamo il registro %eax in cui andremo precedentemente a memorizzare il numero della sys call voluta. Le sys call necessitano anche di argomenti come le normali funzioni; il passaggio avviene tramite i registri; l'ordine degli argomenti e' (%ebx, %ecx, %edx, %esi, %edi, %ebp). Il valore di ritorno e' restituito in %eax.

es:

```
    movl $0, %eax // Sys call N° 0  
    movl $10, %ebx // 1° Arg = 10  
    int $0x80      // SW Interrupt
```

%eax = N° sys call
%ebx = 1° Arg
⋮
%ebp = 6° Arg

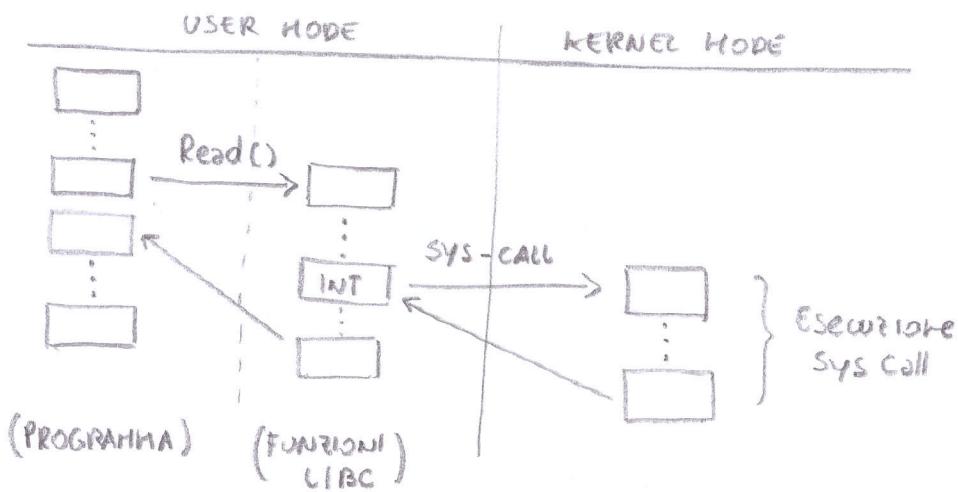
PORATIBILITÀ DEL SOFTWARE

Valore di Ritorno in %eax

Normalmente un programma scritto in C sfruttando la LIBC, e' PORTABILE in tutti i sistemi operativi poiche' tale libreria offre un'interfaccia comune a tutti, con il termine portabile si intende che uno stesso file sorgente C puo' essere compilato su diversi O.S.

Tuttavia i file sorgenti che usano direttamente le system call NON SONO PORTABILI poiche' alcuni sistemi operativi hanno diverse sys-call. Ad esempio la stessa LIBC NON E' PORTABILE, le macro associate alle system call sono descritte in `usr/include/unistd.h`.

Per rendere portabile un sorgente si deve quindi utilizzare la LIBC che contiene i wrapper per le system call in modo che il software sia portabile e solo la libreria dipenda dal O.S.



• STANDARD POSIX (IEEE 1003) (ISO/IEC 9945)

In realtà non tutti i O.S. adottano differenti System call, infatti i sistemi operativi che adottano lo standard POSIX (Portable Operating System Interface for UNIX) hanno le medesime sys-call di base.

La maggior parte dei O.S. adotta tale standard eccetto per Windows e ANDROID che non vi hanno aderito.

Il Posix standardizza diversi aspetti del OS, tra cui:

- System Call
- LIBC: La libreria è portabile
- Core Utils: (es: ls, cd, mv, cp, rm ...)
- Albero delle directory
 - [es:]
 - usr/sbin/ → Programmi del sistema
 - usr/bin/ → Programmi utente
- Bin Utils: Strumenti per file binari (es: as, objdump)

- API (Application Programming Interface) (POSIX)

Abbiamo visto che un processo utente non gestisce direttamente l'hardware della macchina, vi e' infatti il O.S. che fa da intermediario, la parte piu' importante di quest'ultimo e' il Kernel.

Tra le componenti vi sono le interfacce:

- ABI (Application Binary Interface): Definisce l'interfacciamento tra l'hardware ed il Kernel/O.S.
- API (Application Programming Interface): Definisce l'interfacciamento tra il Kernel/O.S. e i processi utenti.

Tramite la LIBC il POSIX standardizza l'API in questo modo un software risulta portabile tra diversi O.S se utilizza la LIBC.

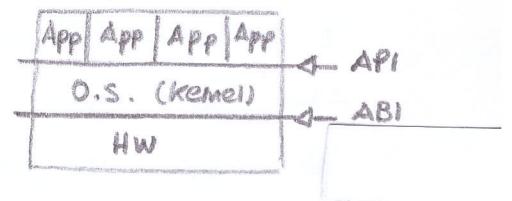
Il POSIX e' gestito da Open Group Posix, nel loro sito sono descritte le specifiche delle diverse componenti, tra cui quindi le funzioni della LIBC. Si osservi che tali specifiche NON si riferiscono all'implementazione ma solo al comportamento ed all'interfaccia. Per determinare esattamente come vengono implementate si utilizza il comando shell MAN, che consente l'accesso al manuale diviso in tre parti:

- SEZIONE 1: Funzioni di Shell
- SEZIONE 2: Wrapper delle system-call
- SEZIONE 3: Funzioni della LIBC

Inserendo come primo argomento il numero della sezione voluta si ottiene la specifica della relativa funzione

>> man 3 printf # Restituisce man di printf della LIBC

Si osservi infatti che printf e' anche un comando della shell, senza la specifica della sezione, la man restituisce la funzione in riferimento alla prima sezione in cui compare



WRAPPER

Le WRAPPER (Rivestimenti) sono funzioni della LIBC, scritte in ASH che eseguono essenzialmente le system-call tramite INT.

Tali wrapper vengono poi utilizzate dalle altre funzioni della LIBC. Ad esempio la funzione WRITE e' un wrapper che esegue la sys-call write, e che viene usata nella PRINTF.

• ESEMPIO (-exit)(-nostdlib)

Vediamo ora come scrivere un wrapper, i codici relativi alle sys-call sono descritti nel header unistd-32.h della LIBC.

Tramite il comando LOCATE possiamo ricavarne il percorso:

>> locate unistd-32.h

Il nostro obiettivo e' implementare il wrapper per la system call exit che consente di terminare il processo in esecuzione.

Lo standard POSIX impone la seguente specifica per il wrapper -exit

void _exit(int status) | man 2 _exit

dove "status" indica lo stato di uscita del processo (exit status)

Nei programmi che compiliamo solitamente la funzione main(). Non e' mai la prima ad essere eseguita, inoltre allo' fine di essa vengono chiamate altre funzioni per la "pulizia" del processo.

La prima funzione ad essere chiamata e' _start e nell'ultima funzione eseguita (dopo il main) una volta terminate le azioni di pulizia viene invocato il wrapper _exit() che termina il processo.

Nel nostro esercizio NON vogliono altre funzioni di prologo ed epilogo
in modo tale da verificare che la nostra sys-call exit sia l'unica
ad essere eseguita

L'unica funzione che contiene il nostro programma e' -start

```
.globl _start          # Komkaze.s
_start:
    movl $1, %eax  # numero sys-call per exit
    movl $0, %ebx  # 1° Arg della sys-call (status=0)
    int $0x80      # sys-call
    ret
```

Si osservi che le wrapper della CIBC hanno i medesimi argomenti
delle sys call, nel nostro caso ad esempio sia il wrapper -exit()
che la sys call exit hanno come argomento il exit status.

Notiamo inoltre che NON c'e' stato bisogno di fare lo push e
la pop del registro %ebx poiche' in questo caso -start e'
l'unica funzione eseguita

Naturalmente NON possiamo compilare normalmente poiche' abbiano
redefinita la funzione -start.

Per compilare senza prologo ed epilogo usiamo il seguente comando:

```
>> gcc -m32 -nostdlib -fPIE --build-id=none -o komkaze Komkaze.s
```

EXIT STATUS

Quando un processo termina esso restituisce al chiamante lo stato di uscita (exit status). Per convenzione lo stato 0 indica che il processo si è concluso correttamente, un valore diverso da 0 indica una terminazione causata da un errore.

La LIBC contiene due macro in stdlib.h che rendono il codice più leggibile

EXIT-SUCCESS 0

EXIT-FAILURE 1 (valore indipendente da POSIX)

LETTURA EXIT STATUS

Un processo per leggere lo exit status di un processo figlio accede a delle variabili dell'ambiente.

Per leggere lo exit status di un processo lanciato da terminale occorre accedere alla variabile di ambiente "?"

In bash:

\$ VAR (lettura variabile VAR)

echo \$VAR (Stampa VAR)

Dunque per leggere l'exit status, ad esempio di kamikaze:

>> ./Kamikaze

>> echo \$?

N.B. (Convenzione exit status)

La convenzione sull'exit status si riferisce ai soli processi e NON ai valori di ritorno delle funzioni in generale. Solo nel caso del main() il valore di ritorno corrisponde all'exit status e quindi viene rispettata tale convenzione.

ERRNO

Nella LIBC c'è presente in errno.h una variabile esterna che viene utilizzata dalle FUNZIONI per memorizzare il codice di errore, in questo modo se una funzione ritorna -1, tale indice in errno nell'esecuzione, in errno c'è presente il codice che specifica l'errore verificato.

Vediamo ora come sfruttare errno invocando ad esempio il wrapper write()

• ESEMPIO

La funzione write() ha la seguente segnatura:

ssize_t write(int fd, const void* buf, size_t count)

dove:

size_t = Tipo che rappresenta una dimensione (unsigned)

ssize_t = Tipo che rappresenta una dimensione come size_t ma che può assumere il valore -1 (signed)

fd = (File Description) Indica il canale in cui scrivere.

buf = Buffer da stampare

count = Numero di byte da stampare

Tale funzione restituisce normalmente il numero di byte scritti, se si è verificato un errore invece restituisce -1, in quest'ultimo caso setta correttamente errno in modo da classificare l'errore.
Noi saremo interessati a seguenti codici di errore:

EAGAIN → Errore File Description

EBADF → File description NON valido

File Description

Il file description (fd) indica il canale di I/O, un canale si può aprire ad esempio verso un file oppure verso un socket di rete.

Al momento dell'esecuzione di un processo sono sempre aperti i seguenti fd:

0 → Standard Input (SI)

1 → Standard Output (SO)

2 → Standard Error (SE)

Dal terminale si possono settare gli standard I/O:

>> ./prog1 < file.txt # prog1 ha come SI file.txt

>> ./prog1 > file.txt # prog1 ha come SO file.txt

>> ./prog1 &> file.txt # prog1 ha come SI e SO file.txt

>> ./prog1 | ./prog2 # prog1 ha come SI l'output di prog2

Di default lo SE coincide con lo SO, per separarli:

>> ./prog1 2> file.txt # prog1 ha come SE file.txt

Quando un processo è lanciato da linea di comando, tutti gli standard di I/O sono il terminale.

perror

Per interpretare correttamente un codice di errore memorizzato in errno, la LIBC mette a disposizione la funzione perror() che, in coda al testo che prende come argomento, aggiunge una stringa che esplicita l'errore verificato.

```
#include <stdio.h>
```

```
void perror(const char*s);
```

Il seguente programma prende in input gli argomenti da passare alla write in modo da poter provare i vari errori

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

int main(int nArg, char* arg[])
{
    if(nArg < 4) return EXIT_FAILURE;
    int fd = atoi(arg[1]);
    char* buf = arg[2];
    size_t count = atoi(arg[3]);
    ssize_t res = write(fd, buf, count);
    if(res == -1)
        switch(errno)
    {
        case EBADF:
            ...
            break;
        case EAGAIN:
            ...
            break;
        default:
            perror("Errore Write:");
            break;
    }
    return EXIT_FAILURE;
}
return EXIT_SUCCESS;
```

Quando un processo e' lanciato tramite terminale: ./prog a1 a2 a3

./prog a1 a2 a3

Al main vengono passati:

char* arg[] = {"./prog", "a1", "a2", "a3"}

int nArg = dimensione arg[]

atoi

La funzione atoi della C/C++ consente di convertire una stringa in un intero.

#include <stdlib.h>

int atoi(const char* nptr);

OSS (Gestore Errori)

La write() classifica diversi tipi di errore (vedi man 2 write), in un programma robusto vanno "switchati" tutti e gestiti correttamente in modo da mantenere una corretta esecuzione.

EXEC

EXEC è una famiglia di funzioni che consentono di trasformare il processo attuale in un altro processo.

```
# include <unistd.h>  
int execv(const char *path, char *const argv[]);  
int execvp(const char *file, char *const argv[]);
```

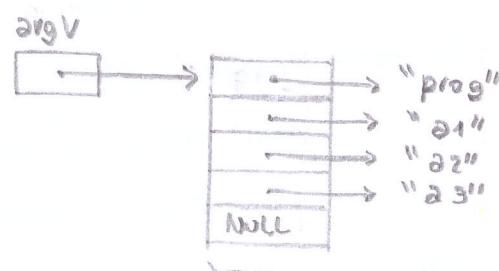
Le due funzioni si distinguono per il fatto che execv necessita del percorso completo del file binario, al contrario execvp ricerca automaticamente il percorso corretto.

OSS

char * const indica che i puntatori agli argomenti sono costanti

Si ricorda che un processo deve ricevere come primo argomento il nome con il quale è stato invocato.

Dato che EXEC NON ha nessun'informazione riguardo alla dimensione dell'array di stringhe, si aspetta come ultimo elemento un puntatore a NULL.



```
#include <unistd.h>  
#include <stdio.h>  
int main()  
{  
    char *const argv[] = {"ls", "-l", NULL}; } // execvp → execv  
    int res = execvp("ls", argv); } // ls → /bin/ls  
    if (res == -1) perror("Error in exec")  
    printf("Arrivo qui solo se exec fallisce");  
    return EXIT_FAILURE
```

OSS (Sovrascrittura del processo)

Quando viene invocata una EXEC il processo attuale viene completamente sovrascritto dal nuovo, quindi EXEC ritorna un valore al chiamante solo se si è verificato un errore (return -1).

Si noti infatti che la stampa finale viene eseguita solo se si è verificato un errore nell'avvio del nuovo processo.

FORK

Notiamo che EXEC ha un limite abbastanza grande poiché non consente di eseguire altre istruzioni dopo la sua invocazione (se andata a buon fine).

In POSIX per eseguire due processi contemporaneamente si utilizza la sys call FORK che DUPLICA il processo attuale andando a copiare la sua intera memoria.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void)
```

Nella duplicazione dei processi si ha una distinzione tra processo padre e processo figlio, la fork() restituisce quindi due valori differenti a seconda del processo.

pid_t pid = fork

pid → = 0 Processo Figlio
→ > 0 Processo Padre

Il tipo pid-t (Process Id Type) rappresenta il numero del processo rispetto all'intero sistema.

I pid partono da 1 e vengono incrementati ad ogni nuovo processo, il pid 0 non esiste

Il valore di ritorno di fork() assume due significati differenti a seconda del ruolo. Nel caso di processo padre, il valore di ritorno e' un vero pid che si riferisce al pid del processo figlio generato.

Il figlio riceve il valore 0 che non rappresenta il pid ma e' solo per distinguere obbligo che nessun processo ha pid=0.

Un processo per ottenere il proprio pid puo' usare la seguente funzione:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void)
```

Inoltre e' possibile conoscere il pid del proprio padre:

```
pid_t getppid(void)
```

• WAIT

Mediente la sys call wait e' possibile attendere la fine del processo figlio generato mediante la fork().

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *wstatus)
```

Tale funzione ritorna il pid del processo figlio una volta terminato, e come argomento prende un puntatore ad intero dove memorizzare l'exit status del figlio

```
:
fork()
:
int exit_status_child;
pid_t pid_child = wait(&exit_status_child);
:
```

N.B. (Separabile Memoria)

I due processi generati mediante la fork() hanno la memoria SEPARATA, sono quindi due processi completamente separati che non condividono memoria (neanche lo heap).

Dunque la modifica di una variabile in un processo NON influenza su quella di un altro.

BUFFERING INPUT/OUTPUT

Le funzioni di alto livello della LIBC (no sys-call) utilizzano un buffer nei canali di input-output in modo da ridurre il numero di accessi alle risorse di I/O che produrrebbero un notevole rallentamento del processo.

es:

write()	// No buffer	} // Bufferizzate
printf		
puts		

Tale fatto puo' portare a comportamenti imprevisti come nel seguente es:

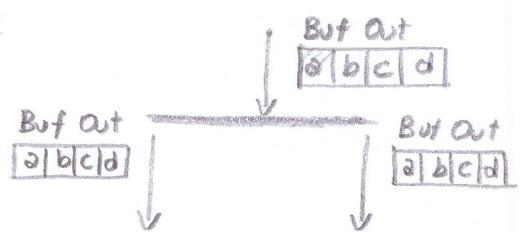
```
// e2-main.c
#include <unistd.h>
#include <stdios.h>
#include <stdlib.h>

int main()
{
    printf("Prima dello fork");
    pid_t pid = fork();
    printf("-dopo la fork\n");
    return EXIT_SUCCESS;
}
```

A differenza di quanto ci aspetteremmo l'output del processo e' il seguente:

```
>> ./e2-main
>> Prima dello fork - dopo la fork
Primo dello fork - dopo la fork
```

Ci aspettavamo infatti che "Primo della fork" fosse stampato solo una volta. La causa di tale comportamento e' il buffer di output usato dalla printf, tale buffer e' memorizzato nel processo utente e quindi nella fork viene duplicato anche tale buffer.



• METODI DI BUFFERING

Il buffer I/O puo' essere svuotato automaticamente dopo determinati eventi oppure essere completamente disabilitato.

Tramite la setvbuf() si puo' impostare un nuovo buffer, moduslib e dimensione:

```
#include <stdio.h>
```

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

dove:

stream = Struttura di tipo FILE che rappresenta il canale di I/O (es: stdin ; stdout ; stderr)

buf = Puntatore al buffer. Se NULL viene utilizzato il buffer di default

mode = Moduli di buffering

size = Dimensione buffer

N.B (setbuf(FILE*, char*))

Tale funzione assume che il nuovo buffer abbia dimensione BUFSIZE

Le moduli di buffering sono le seguenti:

- UNBUFFERED (-IONBF) I/O NON buffered

- LINE BUFFERED (-IOLBF) Il buffer viene svuotato a seguito dell'inserimento del carattere '\n'

- FULLY BUFFERED (-IOFBF) Il buffer viene svuotato una volta pieno
(o BLOCK BUFFERED)

Di default:

LINE BUFFERED se output e' il terminale

FULLY BUFFERED se lo stream e' un file o un socket

Inoltre lo stderr e' UNBUFFERED di default

Per forzare lo svuotamento del buffer si può usare fflush():

```
#include <stdio.h>
int fflush(FILE* stream)
```

PROCEDURA TERMINAZIONE DEL PROCESSO

Abbiamo visto che possiamo terminare un processo tramite la return del main oppure con la exit(), al momento della chiusura vengono eseguite azioni automatiche atte a liberare la memoria precedentemente usata dal processo, tra cui lo svuotamento dei I/O buffer, chiusura dei file. Seppur la chiusura dei file e la deallocazione delle variabili dello heap sono azioni automatiche di chiusura è opportuno includerle direttamente nel codice (fclose(), free())

• ATEXIT

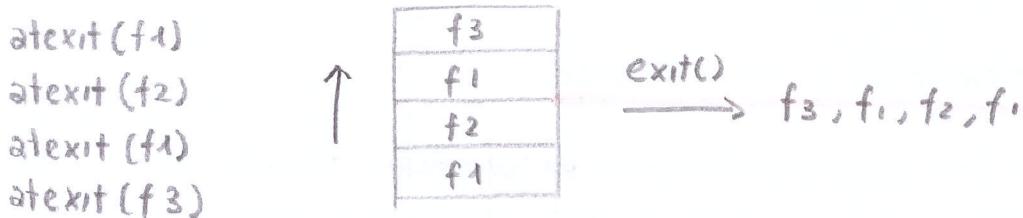
La LIBC mette a disposizione una funzione atexit() che permette di "registrare" alcune funzioni in modo che vengano eseguite al momento della terminazione del processo.

```
#include <stdlib.h>
int atexit(Void (*function)(void))
```

La atexit ha come argomento un puntatore a funzione, può essere usata nel seguente modo:

```
Void bye { printf("Bye\n"); }
int main(){
    :
    atexit(bye);
    :
```

Le funzioni registrate con atexit vengono eseguite al momento della chiusura in ordine inverso a quello in cui sono state registrate.
Si puo' immaginare come una registrazione in pilo:



Si osservi che atexit permette la registrazione multipla di uno funzionaore (f1).

Se si vuole terminare il processo senza eseguire le funzioni registrate tramite atexit() si puo' utilizzare _exit() al posto di exit()

```
#include <unistd.h>
int _exit(int status)
```

OSS (fork())

Come precedentemente visto al momento della fork() la memoria del processo viene duplicata, quindi le funzioni registrate con atexit() prima dello fork() sono registrate sia nel processo padre che in quello figlio.

CONTROLLO DEL FLUSSO

Dal momento dell'accensione fino allo spegnimento del computer, la CPU esegue una serie di istruzioni, tale sequenza viene detta FLUSSO DI CONTROLLO DELLA CPU (CPU's control flow) (CCF). Per alterare tale flusso ci sono due diversi meccanismi:

- VARIAZIONE STATO DEL PROGRAMMA: Normalmente modificano il CCF tramite le istruzioni di salto (call, return) e le condizioni if-else. Tali istruzioni "reagiscono" allo STATO DEL PROGRAMMA.
- VARIAZIONE STATO DEL SISTEMA: Il CCF è modificato anche al momento di eventi imprevisti, come ad esempio l'arrivo di dati dal disco, da rete oppure terminazione forzata di un processo (CTRL-C)

I meccanismi che reagiscono alle variazioni dello stato del programma (call, return, if-else) non sono utilizzabili per reagire correttamente alle variazioni dello stato del sistema, si usa quindi un altro meccanismo detto di INTERRUPT.

FLUSSO DI CONTROLLO ECCEZIONALE

Come precedentemente visto gli interrupt si dividono in due tipi:

- INTERRUZIONE ASINCRONA
- INTERRUZIONE SINCRONA

Interruzione asincrona (interrupt)

Tale interruzione è causata da eventi esterni al processore:

- > I/O interrupt
- > Hard reset interrupt (Tasto Reset)

- Interrupt sincrono

Questi tipi di interruzioni sono dovute ad eventi causati dall'esecuzione di determinate istruzioni nel programma.

Si dividono in tre tipi:

- > Trap: Interruzione volontaria (es sys-call)
- > Fault: Interruzione INNOCENTIALE e può essere recuperabile (es seg fault)
- > Abort: Interruzione INNOCENTIALE e NON recuperabile (es controllo parita')

PROCESSI CONCORRENTI (Concurrent Processes)

Innanzitutto è utile ricordare la differenza tra PROGRAMMA e PROCESSO.

Un processo è un'istanza di un programma a cui garantisce due aspetti fondamentali:

- FLUSSO DI CONTROLLO "ISOLATO": Il programma crede di avere l'intera CPU e suoi dispositivi (NON si accorge dell'esecuzione di altri processi)
- MEMORIA PRIVATA: Il programma crede che l'intera memoria del calcolatore sia a sua completa disposizione, tale illusione avviene tramite il concetto di memoria virtuale

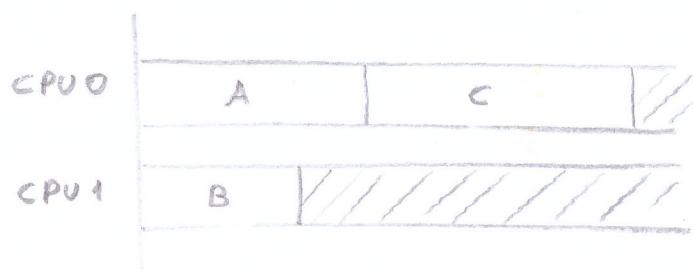
Due processi si dicono CONCORRENTI se il loro flusso di controllo si sovrappongono nel tempo, altrimenti si dicono SEQUENZIALI

es

Supponiamo ad esempio che il calcolatore abbia due CPU:

A, C → Processi sequenziali

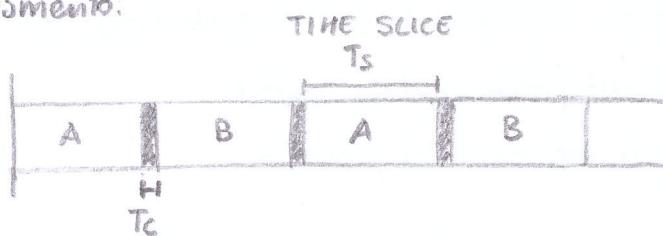
A, B → Processi paralleli



Dal precedente esempio notiamo che NON possiamo eseguire contemporaneamente tutti i processi dato che si necessiterebbe di un'enorme numero di CPU.

Per garantire l'illusione di unicità ad un processo, il kernel simula la concorrenza alternando l'esecuzione dei processi:

Ad esempio l'esecuzione di due processi A, B su single-core ha il seguente andamento:



Il tempo di esecuzione di una porzione di un processo è detto TIME SLICE T_s (fetta di tempo), tra una porzione e l'altra vi è un tempo T_c necessario per il CONTEXT SWITCH.

Il time slice deve avere dimensione contenuta in modo da rendere meglio l'illusione di concorrenza, tuttavia deve avere durata maggiore di T_c . Generalmente T_s è nell'ordine dei millisecondi.

L'ordine di esecuzione e il time slice dei processi viene detto SCHEDULUNG, questo viene deciso dal kernel. Una tecnica fairness (equità) è quella di assegnare ad ogni processo lo stesso T_s e di eseguirli in sequenza circolare, tali strategie è detta ROUND ROBIN.

Nella pratica si usa un metodo che combina la ROUND ROBIN con il concetto di priorità di un processo, in Linux si può varrare la priorità di un processo tramite il comando NICE

PCB (Process Control Block)

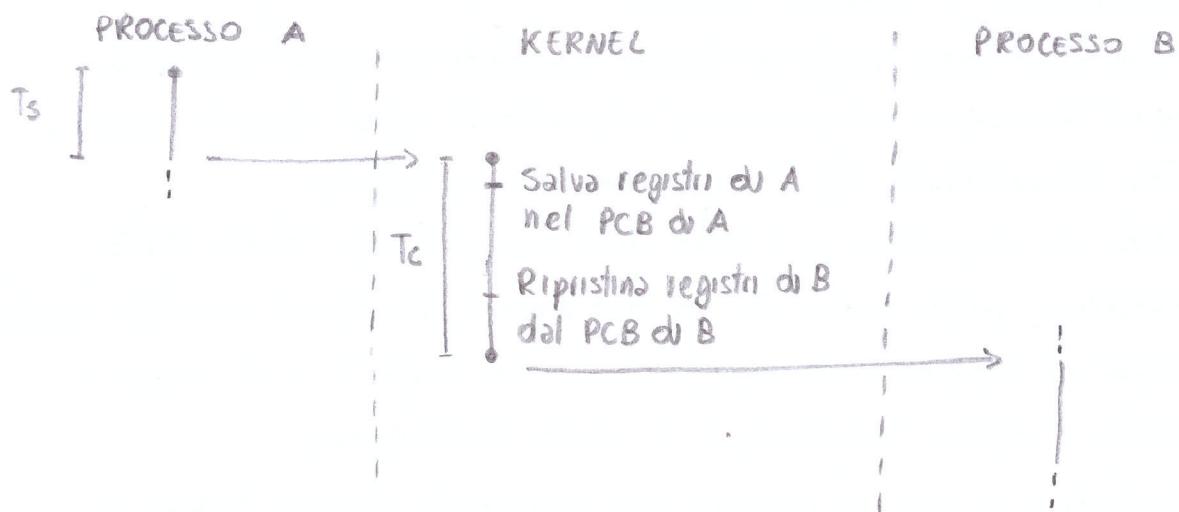
Il Kernel per eseguire un context switch, ovvero il passaggio da un processo A ad uno B, deve memorizzare lo "stato" del processo in modo da ripristinarlo al prossimo time slice.

Le informazioni riguardo un processo sono memorizzate nel relativo PROCESS CONTROL BLOCK (PCB).

Il PCB di un processo contiene diverse informazioni, tra cui:

- Registri CPU dell'ultimo context switch
- puntatori aree di memoria del processo
- PID

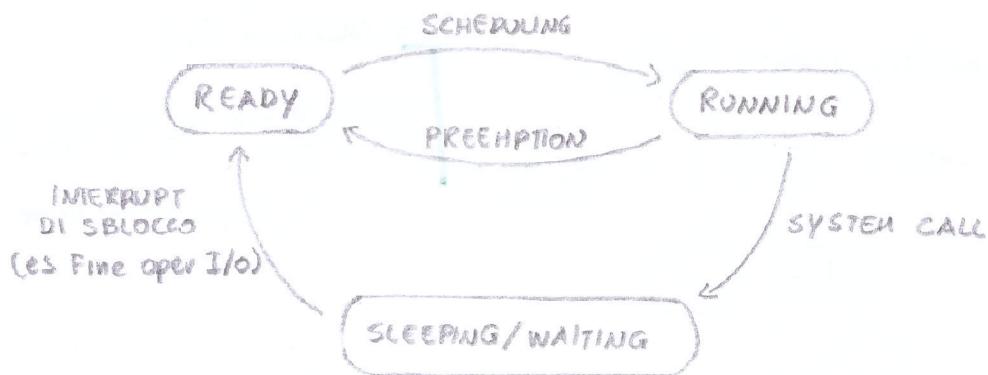
Esempio (Context Switch (A → B))



STATI DI UN PROCESSO

I diversi processi che vengono eseguiti in un moderno Sistema multiprogrammato durante la loro vita passano in diversi stati.

Consideriamo un processo A che genera un processo figlio B, vediamo i possibili stati che puo' attraversare quest'ultimo.



Come visto precedentemente un processo passa dallo stato READY a RUNNING o viceversa a causa dell'esecuzione in parallelo di più processi. Un processo passa nello stato RUNNING se selezionato dallo SCHEDULING, passa invece in READY, ovvero pronto da eseguire ma non in esecuzione, tramite il PREEMPTION, tale meccanismo consente di togliere l'esecuzione ad un processo evitando il fenomeno della STARVATION, dove un processo rimane in attesa perenne senza progredire nella computazione perche' altri processi tengono per se tutti i core del CPU. Lo stato WAITING e' necessario per evitare che la CPU rimanga bloccata in attesa della fine di una system call come ad esempio l'attesa di un pacchetto di rete o la lettura da disco.

Zombie (LEZIONE 14-05)

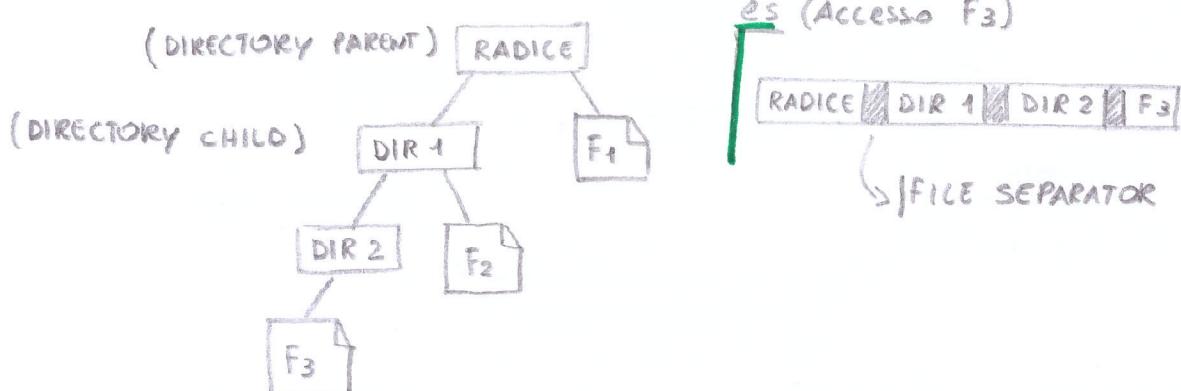
Un ulteriore stato che puo' attraversare un processo e' lo stato ZOMBIE. Un processo puo' terminare in qualsiasi stato visto precedentemente ma alla sua terminazione entra nello stato zombie in cui il processo e' concluso, ma il suo PCB rimane in memoria in modo che il padre possa raccogliere il suo exit status.

Se un processo figlio termina dopo la morte del padre, entra nello stato zombie e dato che non ha piu' un padre viene "adottato" da un altro processo RIPPER che lo uccide leggendo il suo exit status.

FILE SYSTEM

Per archiviare correttamente i dati sul disco rigido si utilizza un FILE SYSTEM che consente di accedere ad un determinato file senza conoscere la sua esatta posizione fisica sul disco.

Un filesystem ha una struttura ad albero:



Riguardo i file system, in POSIX e WINDOWS si hanno due differenze:

- File Separator

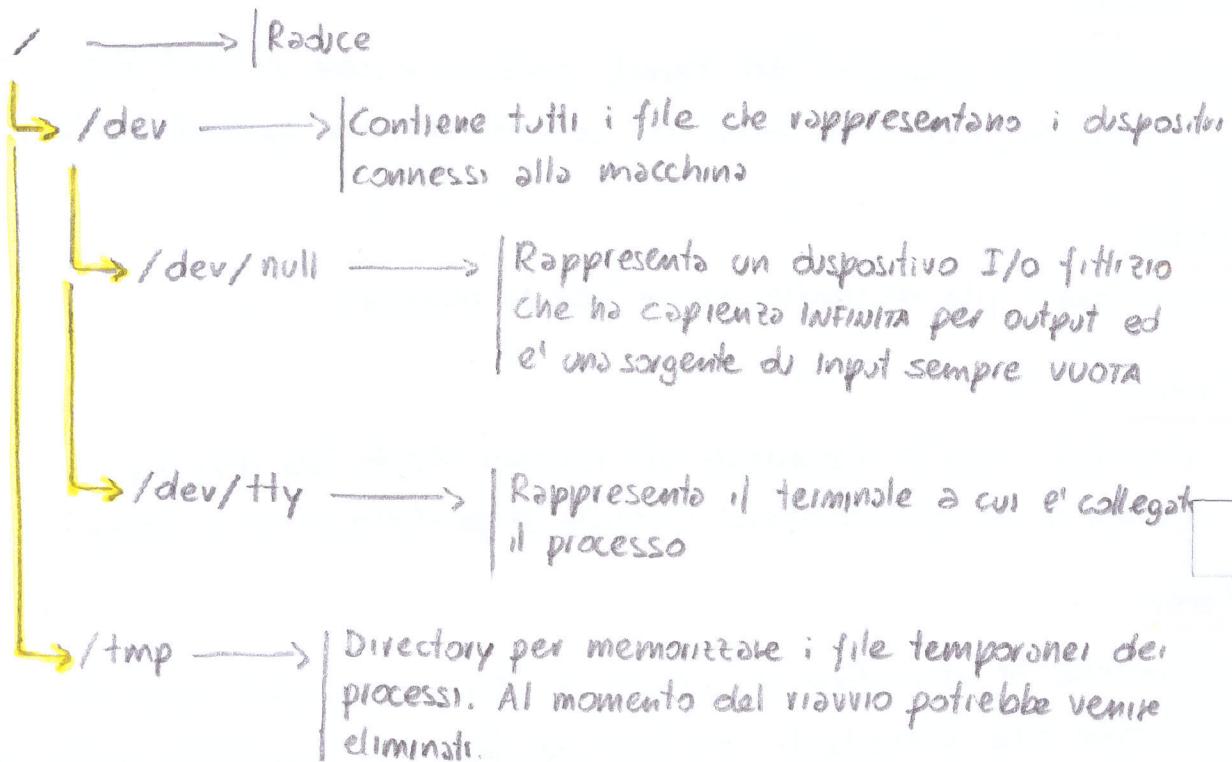
- POSIX → '/'
- WINDOWS → '\\'

- Case Sensitive

- POSIX e' case sensitive → file.txt ≠ FILE.txt
- WINDOWS NON e' case sensitive → file.txt = FILE.txt

- STANDARD POSIX

Lo standard POSIX impone una certa struttura per i filesystem, ovvero devono esserci le seguenti directory:



- STANDARD LINUX (Filesystem Hierarchy Standard) (FHS)

Lo standard Linux FHS estende lo standard POSIX con le seguenti directory:

- /bin

Al momento dell'avvio il sistema è in modalità singolo utente (root) poi diventa multeutente (root, user).

Nella cartella /bin sono presenti tutti i programmi da eseguire in modalità singolo utente.

- /sbin

Contiene i programmi orientati per il sistema operativo (es: mount)

- /boot

Contiene le immagini del kernel, possono essere presenti più versioni del kernel in modo da selezionarne una specifica all'avvio.

- /etc

Contiene i file di configurazione per la macchina

- /home

Contiene i file e documenti del relativo utente. Tale directory è l'unica in cui puoi scrivere l'utente proprietario, oltre a /tmp.

- /usr

Contiene le directory /usr/bin e /usr/sbin che sono le analoghe alle precedenti, ma sono designate per la modalità multi utente.

Naturalmente vi sono anche ulteriori directory.

PATH (variabile di ambiente)

Nella shell bash di Linux esiste una variabile di ambiente che consente di conoscere la posizione dei binari che è possibile eseguire dal terminale, o meglio le directory in cui l'interprete andrà a cercare il comando digitato.

echo \$PATH → Stampa le directory contenente gli eseguibili dal terminale

» ls → L'interprete cerca il binario 'ls' nella directory restituite da \$PATH.

PERCORSO RELATIVO

Quando si indica il percorso di un file partendo dalla radice si dice che il percorso è **ASSOLUTO**.

Il percorso di un file può anche essere **RELATIVO** rispetto ad uno determinato directory.

Al momento dell'esecuzione di un processo, ad esempio lo stesso terminale, la directory in cui si sta lavorando è detta **Process Work Directory (PWD)**, tramite il comando `pwd` si ottiene il percorso assoluto di tale cartella:

```
>> pwd  
/home/NAME-UTENTE
```

Per indicare percorsi RELATIVI si possono utilizzare i seguenti simboli:

- → Indica la directory corrente (`pwd`)
- .. → Indica la directory padre di `pwd`
- ~ → Indica la directory `/home/NAME-UTENTE`

Tali simboli vengono sostituiti con il corretto percorso automaticamente dalla shell in modo da ottenere il percorso assoluto.

SORGENTE RANDOM (LEZIONE 16-05)

Lo standard Posix include anche due sorgenti di dati random:

`/dev/random` → Sorgente random che si blocca se i dati generati non hanno una sufficiente entropia

`/dev/urandom` → Sorgente random che non si blocca

Per generare file binari randomici possiamo usare il seguente comando

```
>> dd if=/dev/urandom of=file.txt count=8192 bs=6096
```

Numero Blocco Dimensione Blocco

COMANDI CORE UTILS

Vediamo ora i comandi più importanti utilizzabili su terminale Linux.

- >> **pwd** // stampa la directory corrente
- >> **cd DIR** // Cambia directory
 - // cd.. Porta alla directory padre
- >> **ls DIR** // Mostra il contenuto della directory specificata
 - // ls -a Mostra i file nascosti (es .file.txt)
 - // ls -l Mostra le specifiche dei file
 - // ls -h Rende l'output Human readable
 - // ls -alh Combina le precedenti opzioni
- >> **touch FILE** // Aggiorna la data di ultima modifica del file specificato
 - // Se non presente crea un nuovo file
- >> **mkdir DIR** // Crea una nuova directory
- >> **rmdir DIR** // Elimina la directory DIR che deve essere vuota
- >> **rm FILE** // Rimuove il file specificato
 - // rm -r DIR Consente di eliminare una cartella non vuota.
- >> **mv S D** // Sposta la sorgente S (FILE/DIR) nella destinazione D
- >> **cp FILE DIR** // Sposta FILE nella directory DIR
 - // cp -r DIR1 DIR2 Copia l'intero cartella DIR1 in DIR2
- >> **hexdump FILE** // Stampi in esadecimale il contenuto del file.

N.B. (rm, rmdir)

I comandi rm, rmdir eliminano definitivamente le directory o i file, senza spostarli nel cestino.

PERMESSI DEI FILE

In Linux un file puo' essere manipolato in diversi modi a seconda dei permessi che ha l'utente utente su quel file.

In riferimento ai permessi si distinguono tre figure:

- PROPRIETARIO (User) : Proprietario del file
- GRUPPO (Group) : Gruppo a cui appartiene il proprietario del file
- ALTRI (others) : Utenti esterni al gruppo.

Per conoscere i permessi che si hanno su un file o directory si usa

» ls -alh

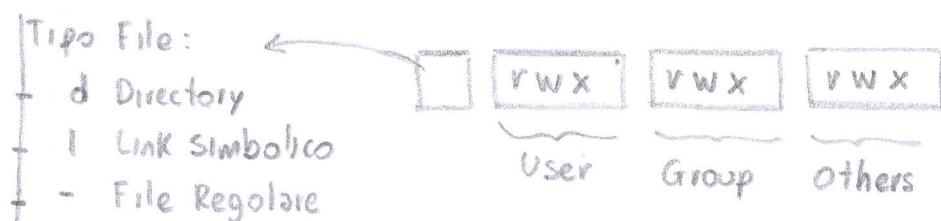
L'output ha un aspetto simile al seguente:

drwxr-xr-x	7	gio	gio	4.0K	april 26 21:32	Documenti
-rw-r--r--	1	gio	gio	807	february 26 15:50	.profile

Permessi Proprietario Dimensione file (No dimensione directory) Nome directory o file

Data ultima modifica

La colonna dei permessi ha il seguente formato:



A seconda del tipo del file i permessi (r w x) assumono significati differenti

FILE	DIRECTORY
r → Readable	r → Readable (lettura contenuti)
w → Writable	w → Writable (Gestione contenuti)
x → Executable	x → Navigabile

I singoli blocchi dei permessi possono essere espressi in binario ponendo il bit a 1 se si ha quel determinato permesso, 0 altrimenti

$$\boxed{rwx} = 111(7) \quad \boxed{r--} = 100(4) \quad \boxed{rw-} = 110(6)$$

Tali blocchi possono essere espressi anche in cifre ottali, queste sono appunto composte da tre bit (0 - 7).

Un numero in cifre ottali e' sempre preceduto da zero, i permessi su un file possono essere rappresentati nel seguente modo

$$\underbrace{rwx}_{7} \underbrace{rwx}_{7} \underbrace{r-x}_{5} = 11111101 = 0775$$

chmod
chown

HANIPOLAZIONE FILE IN C

Vediamo ora i wrapper usati nella LIBC per gestire un file, le operazioni che possono essere eseguite sono le seguenti:

- APERTURA
- CHIUSURA
- LETTURA
- SCRITTURA

Tali operazioni sono valide non solo su file, ma su qualsiasi canale di comunicazione (es terminale, socket) si basano infatti sul concetto di FILE DESCRIPTOR.

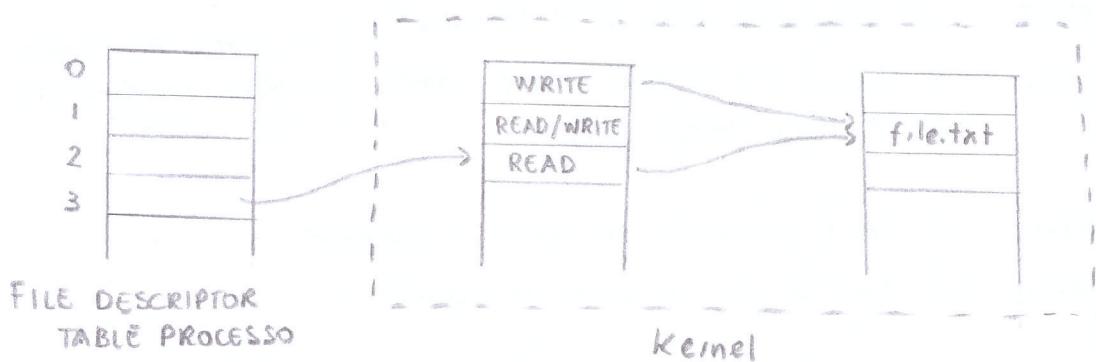
• FILE DESCRIPTOR

Per ogni processo il Kernel genera una FILE DESCRIPTOR TABLE che rappresenta i canali (stream) aperti dal relativo processo, i primi tre stream sono sempre gli stessi e sono aperti automaticamente.

0	stdin
1	stdout
2	stderr
3	

Un file descriptor rappresenta un indice della precedente tabella, è possibile vedere i fd di un processo nello directory /proc/PID/fd dove PID è l'identificativo del processo. (PID = getpid())

A livello di Kernel si ha anche una tabella, unica per tutti i processi, la quale rappresenta i file aperti e la relativa modalità.



- lsof

Da terminale e' possibile vedere tutti i file aperti dal kernel, usando il seguente comando:

>> lsof

Per vedere quale processo ha aperto uno specifico file filename.txt possiamo utilizzare grep:

>> lsof | grep filename.txt

Si osservi che editor di testo come Geany non tengono aperti i file, ma vi accedono solo quando necessario

• APERTURA

Il wrapper usato per l'apertura di un file e' open():

int open(const char *pathname, int flags)

int open(const char *pathname, int flags, mode_t mode)

Flags

L'argomento FLAGS specifica la modalita' di apertura del file, si possono combinare le seguenti opzioni tramite l'or bitwise | :

O_RDONLY // Solo lettura

O_WRONLY // Solo scrittura

O_RDWR // Lettura e Scrittura

O_CREAT // Se il file NON esiste lo crea

O_EXCL // Se il file esiste open() dà errore
// restituisce errore (-1)

O_TRUNC // Se il file esiste ne tronca il contenuto

O_APPEND // Apre il file e scrive in append

O_CLOEXEC // Chiude automaticamente il file.

dove:

pathname = Percorso del file

flags = Modalita' di apertura

mode = Permessi nuovi file

} Deve sempre essere presente uno di questi

} O_EXCL ha senso solo in combinazione con O_CREAT

- Mode

L'argomento mode, se il file e' aperto con flag contenente O_CREAT, rappresenta i permessi da attribuire al nuovo file creato.



HACR & OTTAH

◦ CREAT

◦ READ

↳ NC lorem.txt

WRITE



CLOSE

|seek()

|tell()

LSEEK()

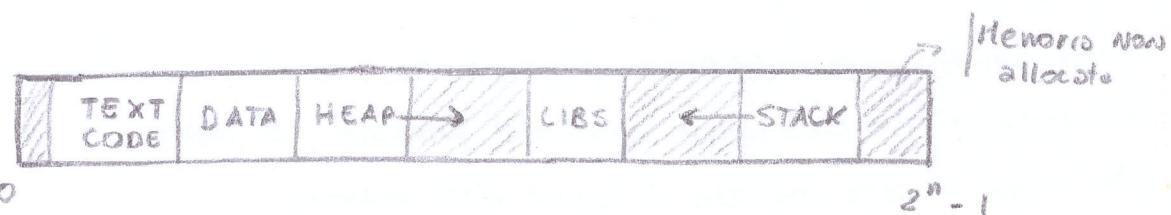
FNUC()

MEMORIA VIRTUALE

Uno degli aspetti più importanti per un sistema multiprocesso è il concetto di **MEMORIA VIRTUALE** che illude i processi di avere a disposizione l'intera memoria del calcolatore.

• ORGANIZZAZIONE MEMORIA DI UN PROCESSO

L'intera memoria del processo è organizzata nel seguente modo:



Si osservi che la memoria in un architettura ad n bit è indirizzata da 0 a $2^n - 1$.

Vediamo le singole parti:

- **O (NULL)**: Tale indirizzo NON è valido in modo tale che l'indirizzo NULL (0) NON assenca a memoria allocata.

- **TEXT CODE**: Contiene il codice binario del programma.

- **HEAP**: Memoria heap, tale memoria cresce verso gli indirizzi alti.

- **LIBS**: Contiene il codice binario delle librerie.

- **STACK**: Memoria stack, cresce verso indirizzi bassi.

- **DATA**: Contiene le variabili globali e statiche:

↳ BSS

↳ Variabili Globali /
Statiche
NON inizializzate
vengono fatte a 0

Suddivisione DATA

• ACCESSO MEMORIA DI UN PROCESSO

Nei sistemi operativi Unix-like possiamo navigare i dati dei processi nel percorso "/proc", tale directory è gestita dal Kernel e contiene una sotto-directory per ogni processo, i nomi delle cartelle corrispondono ai PID dei relativi processi.

La directory /proc non è salvata effettivamente sul disco, ma è nella memoria temporanea.

Nella cartella /proc/<PID> possiamo trovare una directory "fd" associata ai file descriptor aperti dal processo ed, inoltre, il file "maps" il quale contiene il layout della memoria.

- Lettura PID

Abbiamo visto precedentemente che un processo può ottenere il proprio PID tramite la funzione getpid().

Per conoscere il PID dei processi lanciati da terminale, possiamo usare il comando di shell "ps" che oltre a fornire i PID, restituisce anche altre informazioni, come ad esempio lo stato del processo.

Si hanno due diverse sintassi per gli eventuali argomenti:

In entrambe le sintassi è possibile unire gli argomenti:

>> ps -elf
>> ps -e -l -f } equivalenti

	Posix	BSD
TUTTI I PROCESSI	-e	-q
LONG INFO	-l	-u
HORE INFO	-f	-x

Il comando ps senza l'opzione -e oppure -q, restituisce solo i processi lanciati dalla medesima shell.

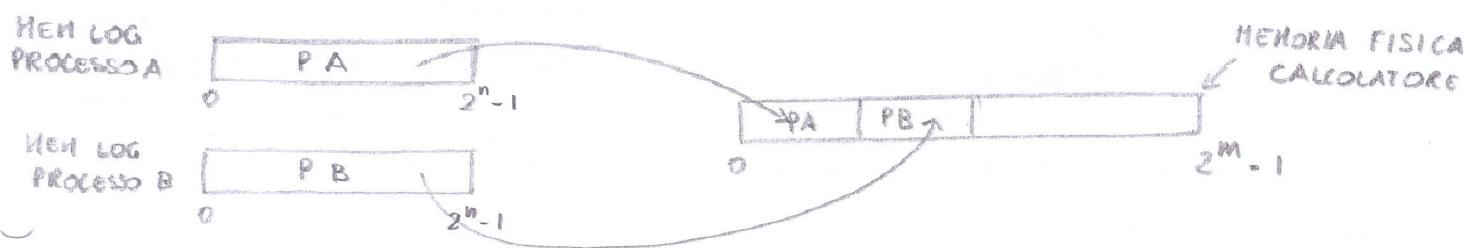
Dato che ps finisce anche il comando con cui è stato lanciato un processo possiamo intracciare tale processo con grep:

>> ./e1 & // & serve per mandare il processo in background
>> ps -elf | grep "./e1"
>> pgrep e1 // come ps | grep, di default restituisce solo il PID

MEMORIA LOGICA E MEMORIA FISICA

La memoria LOGICA o virtuale è quello a cui ci riferiamo nel processo ed è suddivisa in varie parti come visto precedentemente, la memoria FISICA invece è quella effettiva del calcolatore.

Prendiamo in considerazione due processi A e B, le loro memorie virtuali devono necessariamente offrire alla memoria fisica:

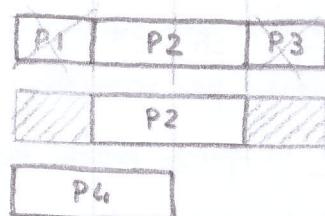


L'assegnazione tra memoria virtuale e memoria fisica è detto HAPPING.

• PAGINAZIONE DELLA MEMORIA

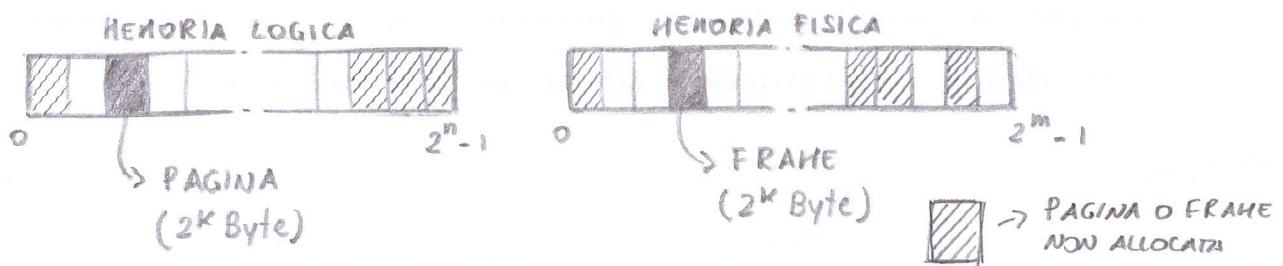
Il mapping della memoria virtuale su quello fisica NON è un meccanismo molto scontato, prendiamo il seguente esempio:

Consideriamo la memoria mappata da tre processi di dimensione diversa, in seguito alla terminazione di P1 e P3 si ha spazio sufficiente per un nuovo processo P4, tuttavia Non si dispone di un area di memoria continua, quindi NON risulta possibile mappare P4.



Un altro problema è dato dal fatto che l'indirizzamento della memoria virtuale di un processo può superare l'effettiva dimensione della memoria fisica ($n > m$), ad esempio in un architettura a $n = 32$ bit la memoria virtuale di un singolo processo è formata da 2^{32} Byte = 4GB quindi un computer con 2 GB di ram non potrebbe mappare neanche un processo.

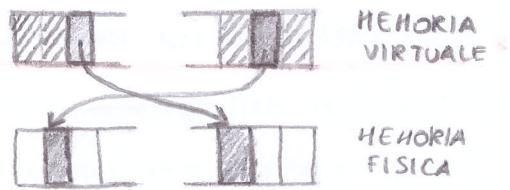
Per ovviare ai precedenti problemi si suddivide la memoria virtuale e fisica in blocchi di memoria detti rispettivamente PAGINE e FRAHE, entrambi i blocchi hanno lo stesso dimensione di 2^k Byte, solitamente $k = 12$ (2^{12} Byte = 4 KB)



In questo modo dividiamo la memoria logica in $2^n / 2^k = 2^{n-k}$ pagine e quella fisica in 2^{m-k} frame.

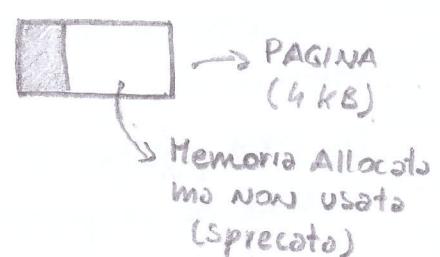
Il mapping quindi avverte un associazione tra pagina e frame, non allocchiamo dunque interi processi.

Tramite mapping PAGINA \rightarrow FRAHE eliminiamo il problema della frammentazione ESTERNA dei processi per cui non era possibile allocare un processo per mancanza di memoria contigua.



Tuttavia in questo modo nasce il problema non risolvibile detto di frammentazione INTERNA, ovvero dato che è possibile allocare solo INTERE pagine (es 4KB) potrebbe accadere che :

Su 4KB solo una piccola parte vengono effettivamente utilizzati, causando quindi uno spreco della memoria. Per ridurre tale problema occorre una dimensione ridotta per le pagine.



• THRASHING

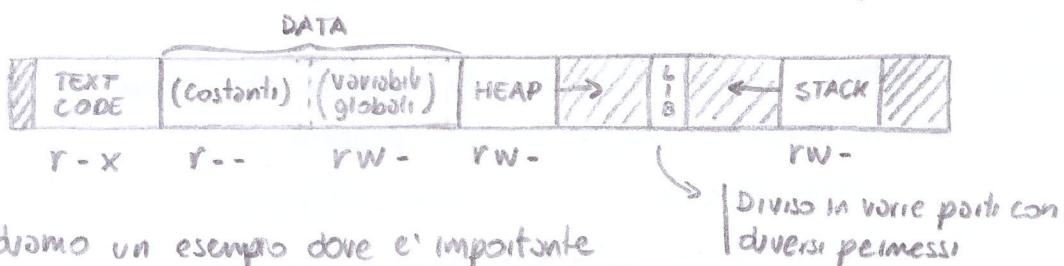
Quando il numero delle pagine, correntemente utilizzate da tutti i processi, e' superiore al numero di frame di cui e' composta la memoria fisica, si ha continuamente una procedura di spostamento di pagine da e verso l'area di swap.

Tale situazione porta ad un rallentamento enorme di tutte le attività del calcolatore, questo fenomeno e' detto THRASHING e si puo' risolvere solamente tramite la terminazione di processi o incrementando la memoria RAM montata sul computer.

PERMESSI SULLE PAGINE

Una pagina, oltre ad essere privata o condivisa, puo' avere diversi permessi (lettura, scrittura, esecuzione).

Nel file /proc/<PID>/maps possiamo vedere tutti i permessi associati alle singole pagine, espressi nel formato rwx p/s (p=private; s=shared)



Vediamo un esempio dove e' importante conoscere i permessi di una pagina:

es:

```
int i = 0; // Variabile globale (rw-)
int main() {
    char s[] = "hello"; // Variabile di tipo array di char memorizzato in stack
    s[0] = 'x'; // OK stack e' rw-
    char* p = "hello"; // p punta alla stringa costante memorizzata
                        // nel blocco DATA costante (r--)
    p[0] = 'x'; // Segmentation fault il blocco DATA costante NON
                // puo' essere scritt.
```

ALLOCAZIONE PAGINE

Vediamo ora come allocare nuove pagine nella memoria virtuale di un processo tramite la system call `mmap()`:

`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`

- `void *addr` = Indirizzo da cui iniziare l'allocazione
(Se `addr = NULL`, il Kernel sceglie un indirizzo opportuno)
- `size_t length` = Dimensione dello spazio di memoria da allocare
(Dere essere un multiplo della dimensione di una pagina)
(`getpagesize()` restituisce la dimensione di una pagina)
- `int prot` = Protezione per la pagina `rwx`, si possono combinare le seguenti macro tramite l'or bitwise (|):
`PROT_EXEC | PROT_READ | PROT_WRITE (rwx)`
`PROT_NONE (Nessun permesso) (---)`
- `int flags` = Determina se la nuova zona di memoria deve essere privata oppure condivisa tra altri processi
`MAP_PRIVATE` `MAP_SHARED`
In aggiunta ad una delle precedenti macro occorre includere anche la seguente macro quando la nuova regione NON riferisce a nessun file descriptor. (`MAP_ANONYMOUS`)
- `int fd` = Se diverso da -1 indica il file descriptor di un file da mappare direttamente nella memoria virtuale del processo
- `off_t offset` = Offset da cui iniziare a mappare il file indicato da `fd`, nel caso sia $\neq -1$.

Se la nuova zona di memoria NON mappa nessun file occorre mettere la flag `MAP_ANONYMOUS`, e porre `fd = -1` e `offset = 0`.

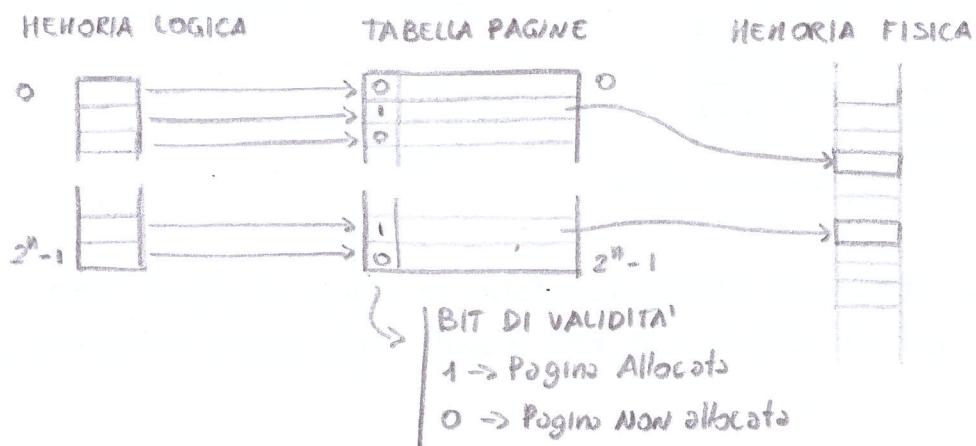
N.B.

`mmap()` NON mappa nuove pagine nello heap, ma in qualsiasi zona NON allocata della memoria virtuale del processo.

- Tabella delle pagine

Il problema di non poter mappare l'intera memoria logica di un processo sulla memoria fisica si risolve semplicemente allocando in memoria solo le pagine necessarie, in questo modo la dimensione della memoria mappata è enorimamente minore di quella totale.

Il sistema operativo memorizza il mapping tramite la **TABELLA DELLE PAGINE**, che distingue tra le pagine allocate da quelle non allocate.



VANTAGGI MEMORIA VIRTUALE

I vantaggi nell'utilizzo di memoria virtuale invece di quella fisica sono dunque i seguenti:

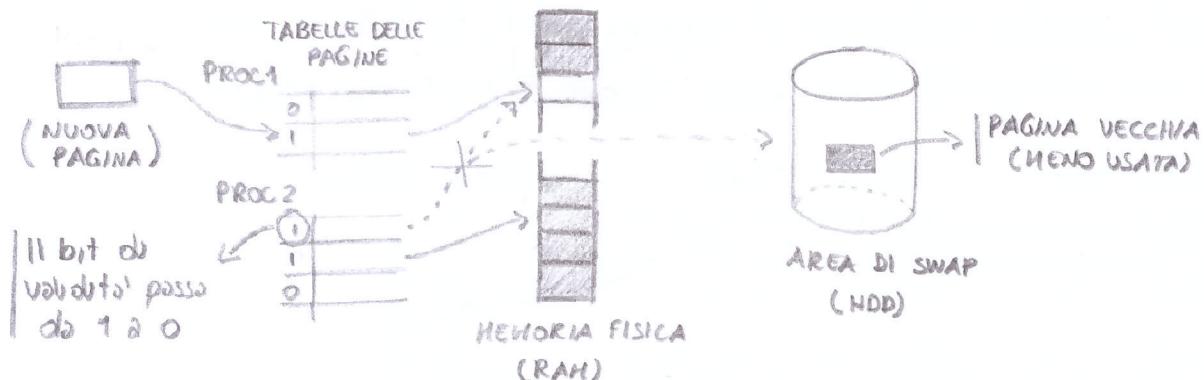
- PROTEZIONE: Ogni processo crede di essere solo in memoria, tale meccanismo garantisce anche l'**ISOLAMENTO** e la possibilità di gestire i **PERMESSI** sulle aree di memoria.

- CONDIVISIONE: Tramite il meccanismo del mapping è possibile condividere una zona di memoria fisica tra più processi, ovvero due zone di memoria virtuale di due o più processi assegnano alla medesima zona di memoria fisica.

- MEMORIA ILLIMITATA: In questo modo si può avere più memoria logica di quella fisica, inoltre il meccanismo di SWAP aumenta maggiormente la capacità.

AREA DI SWAP

Al momento dell'allocazione di un processo, o di una semplice pagina, potrebbe accadere che non ci sia più spazio nella memoria fisica, se ciò accade il Kernel provvede a liberare i frame necessari scegliendo quelli meno utilizzati e spostandoli sul disco rigido in un'area detta di SWAP.



Il Kernel una volta liberata la memoria spostando in swap le pagine meno utilizzate, mette il bit di validità a 0 nella tabella delle pagine corrispondente

• PAGE FAULT

Se un processo, durante un accesso a memoria trova il bit di validità a 0 viene provocato un PAGE FAULT.

Il bit di validità può essere a 0 per due motivi:

- PAGINA NON ALLOCATA
- PAGINA SPOSTATA IN SWAP

Nel primo caso avviene un segmentation fault poiché la pagina a cui si fa riferimento non è effettivamente stata allocata. Nell'altro caso invece il kernel provvede a spostare la pagina richiesta dallo swap alla memoria RAM e intanto l'accesso

Se l'operazione di mapping va a buon fine, mmap() restituisce un puntatore alla nuova zono di memoria, altrimenti la macro MAP_FAILED.
es (Allocazione singola pagina)

```
#include <sys/mman.h> // mmap  
#include <stdio.h> // perror  
#include <stdlib.h> // macro EXIT  
#include <errno.h>  
  
int main(){  
    char *p = mmap(NULL, getpagesize() * 1, PROT_READ | PROT_WRITE,  
                  MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);  
    if(p == MAP_FAILED){  
        perror("Errore mmap");  
        exit(EXIT_FAILURE);  
    }  
}
```

• AGGIORNAMENTO PERMESSI

Le pagine allocate tramite mmap() possono subire variazioni sui permessi mediante la funzione mprotect().

```
int mprotect(void* addr, size_t len, int prot)
```

- void* addr = Indirizzo dell'area di memoria su cui variare i permessi
- size_t len = Dimensione area di memoria
- int prot = Nuovi permessi

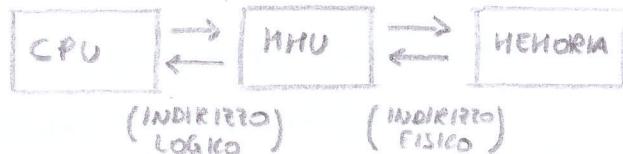
Tale funzione in caso di successo restituisce 0, altrimenti -1.

Si osservi che mprotect() ha effetto solo su zone mappate tramite mmap()

MAPPING

La procedura di MAPPING come visto precedentemente consiste nell'assegnare un indirizzo della memoria virtuale ad uno fisico della memoria del calcolatore.

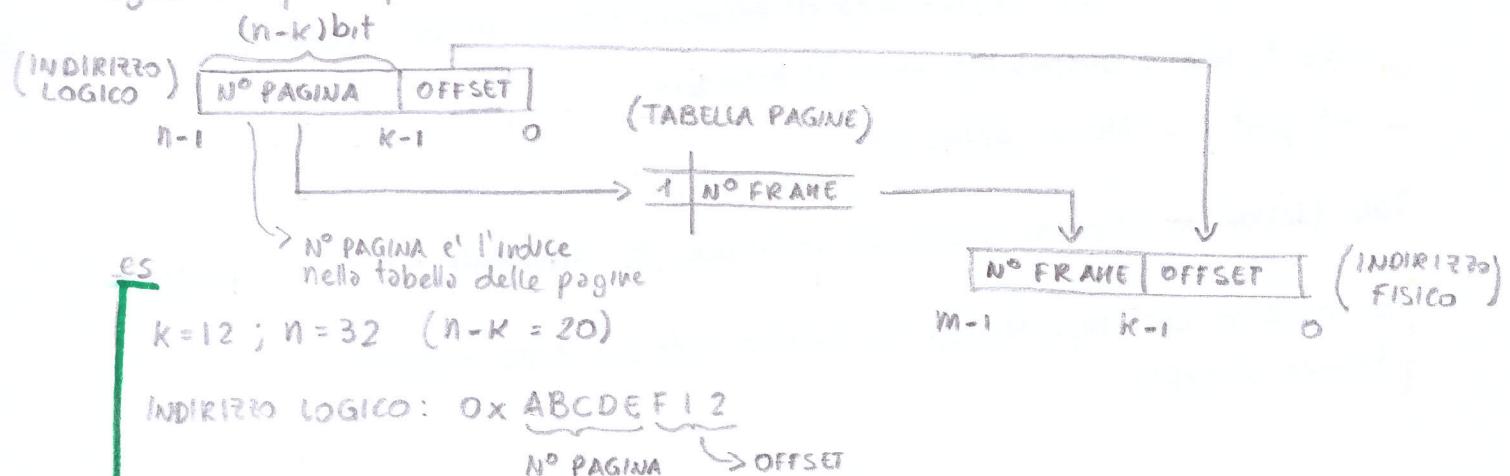
La mappatura viene eseguita da una classe di dispositivi hardware che formano la Memory Management Unit (MMU), tale procedura risulta quindi nascosta al programmatore che non accede direttamente alla memoria fisica.



CONVERSIONE INDIRIZZI

Vediamo come avviene la conversione da indirizzo virtuale a fisico da parte della MMU.

In un architettura a n bit con pagine di dimensione 2^k si ha che la memoria virtuale è suddivisa in 2^{n-k} pagine, quindi per identificare tutte le pagine si necessita di $n-k$ bit. Per selezionare un singolo byte in una pagina si necessita di 2^k bit che chiameremo di offset. Dato che sia i frame che le pagine sono composte da 2^k byte si deduce che l'offset rimane invariato e che quindi il mapping consiste nell'indirizzare una delle 2^{n-k} pagine ad uno dei 2^{m-k} frame, si ricorda che necessariamente si deve avere $n \geq m$. Il mapping avviene nel seguente modo, dividendo in due parti sia l'indirizzo logico che quello fisico.



SEGNALI

Un SEGNALE e' un messaggio destinato ad un processo, puo' essere inviato da altri processi, dal Kernel o da se stesso, tale messaggio non contiene informazioni, ma e' lui stesso ad avere un significato in base al tipo di segnale, o meglio al numero associato.

Un segnale e' definito da:

- Un NUMERO che lo identifica univocamente
- Il PID del processo destinatario

N.B. (Differenza tra interrupt e segnali)

- . Occorre notare la differenza tra interrupt (eccezioni) e segnali.
I primi sono messaggi inviati dal hardware a differenza dei segnali i quali sono inviati solo da figure software (processi / Kernel)

Tramite alcune sys-call e' possibile trarre piu' informazioni dal segnale che caratterizzano lo stato del sistema al momento dell'invio

COMPORTAMENTI PREDEFINITI

I principali comportamenti predefiniti (default) di un processo al momento della ricezione di un segnale sono i seguenti:

- TERM: Il processo ricevente termina

- CORE: Il processo ricevente termina e genera un file chiamato CORE DUMP che consiste in un'istanza dello stato della memoria del processo e dei registri al momento della ricezione.

Il core dump generato puo' essere utile a scopo di debugging

- IGN: Il segnale viene ignorato e non ha alcun effetto sul processo ricevente.

PRINCIPALI SEGNALI

Vediamo ora i segnali più noti suddividendoli in diverse categorie.

• SEGNALI DI TERMINAZIONE

SEGNALE	N°	DESCRIZIONE	IGNORABILE	GESTIBILE	DEFAULT
SIGTERM	15	Terminazione "morbida" in cui il processo ha la facoltà di liberare le proprie risorse prima di terminare	SI	SI	Term
SIGINT	2	Come SIGTERM, ma attivabile anche da tastiera (CTRL+C)	SI	SI	Term
SIGQUIT	3	Come SIGTERM, ma attivabile anche da tastiera (CTRL+\)	SI	SI	Core
SIGKILL	9	Terminazione forzata a cui il processo NON puo' opporsi e quindi NON puo' liberare le risorse	NO	NO	Term
SIGCHLD	17	Segnale che indica la terminazione di un processo figlio.	SI	SI	Ign

Come vedremo successivamente qualora un segnale è gestibile, allora si puo' definire un handler che esegue azioni differenti a quello di default.

Per attendere la fine di un processo figlio possiamo utilizzare un handler di SIGCHLD al posto della funzione wait(), il vantaggio è che in questo modo il padre non resta bloccato in attesa, ma puo' svolgere altre operazioni, il segnale SIGCHLD sarà gestito in modo asincrono.

• SEGNALI ACCESSO A MEMORIA

SEGNALE	N°	DESCRIZIONE	IGNORABILE	GESTIBILE	DEFAULT
SIGSEGV	11	Accesso errato a memoria da parte del processo stesso (Segmentation fault)	SI	SI	Term

Cosa si intende per ignorable?

Gestibile (OK)

• SEGNALE ERRORE ARITMETICO

SEGNALE	N°	DESCRIZIONE	IGNORABILE	GESTIBILE	DEFAULT
SIGFPE	8	Errore aritmetico (es divisione per zero)	SI	SI	Core

Collegamento con interrupt di errore aritmetico?

• SEGNALI PERSONALIZZABILI

SEGNALE	N°	DESCRIZIONE	IGNORABILE	GESTIBILE	DEFAULT
SIGUSR1	10	Segnale liberamente gestibile e utilizzabile dall'utente	SI	SI	Term
SIGUSR2	12	Segnale liberamente gestibile e utilizzabile dall'utente	SI	SI	Term

Tali segnali possono essere usati per scambiare notifiche personalizzate tra diversi processi.

• SEGNALI DI TIMING

SEGNALE	N°	DESCRIZIONE	IGNORABILE	GESTIBILE	DEFAULT
SIGALRM	14	Segnale che indica un evento temporale, come ad esempio un timer	SI	SI	Term

Tale segnale se gestito può essere usato per eseguire una determinata operazione con cadenza regolare

• INVIO SEGNALI

Possiamo inviare diversi tipi di segnali ad un altro processo tramite determinate system call, noi vediamo la "Kill".

La funzione Kill è utilizzabile sia tramite wrapper C, sia direttamente da terminale.

• WRAPPER C

```
#include <signal.h>
int kill(pid_t pid, int sig)
```

- pid.t pid : Se >0 indica il pid del processo destinatario.

- int sig : Segnale da inviare tra quelli definiti in signal.h

Se sig=0 non invia alcun segnale, ma verifica solo la validità del PID destinatario.

Restituisce 0 se l'operazione è andata a buon fine, altrimenti -1

pid ≤ 0 ?

process group

• COMANDO DA TERMINALE

Alla sys call kill corrisponde l'omonimo comando da terminale kill

- >> kill -segnale pid
- >> kill -s segnale pid

I seguenti comandi sono equivalenti:

- >> kill
- >> kill -s SIGTERM 5221
- >> kill -s 15 5221
- >> kill -SIGTERM 5221
- >> kill -15 5221

$$| \text{SIGTERM} = 15$$

GESTIONE SEGNALI (sigaction)

I segnali etichettati come gestibili, possono essere appunto gestiti tramite una funzione handler che viene chiamata al momento della ricezione del segnale.

Per registrare un handler si usa la funzione sigaction

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oldact)
```

- dove:
- int sig: Numero del segnale da gestire; fra quelli listati in signal.h
 - Const struct sigaction *act: Nuova struttura che caratterizza il comportamento del processo al momento della ricezione.
 - struct sigaction *oldact: Struttura attualmente usata la quale viene memorizzata per un eventuale ripristino.

Se l'operazione e' andata a buon fine sigaction() restituisce 0
altrimenti -1.

STRUCT SIGACTION

La struct sigaction usata per definire il comportamento allo ricezione di un segnale ha i seguenti attributi: (ne studiamo solo due)

- void (*sa_handler)(int signum): Gestore del segnale, che può essere una delle seguenti variaz:

- SIG_IGN : Il segnale viene ignorato
- SIG_DFL : Viene utilizzato il gestore da default
- Puntatore ad una funzione definita dall'utente, come ad esempio void handler(int sig) il cui argomento indica il segnale da gestire

• ESEMPIO (Gestore SIGINT)

```
#include ...
void handler(int sig) { printf("Catturato %d dal pid %d", sig, getpid()); }
int main()
{
    struct sigaction act = {0}; // Inizializza la struct
    act.sa_handler = handler;
    if (sigaction(SIGINT, &act, NULL) == -1)
        perror("errore sigaction");
        exit(EXIT_FAILURE);
    while(1); // Mantiene vivo il processo
    return EXIT_SUCCESS;
}
```

N.B. (printf in handler)

Per lo standard POSIX NON dovrebbe essere presente la printf nella funzione handler.

ATTESA SEGNALI

Un processo puo' ricevere segnali in qualsiasi istante, quindi in maniera asincrona rispetto al flusso di controllo.

Un processo puo' anche mettersi in attesa di un qualsiasi segnale tramite la funzione pause():

```
#include <unistd.h>
int pause();
```

Tale funzione sospende indefinitamente il processo fino all'eventuale arrivo di un segnale, dopo di esso il processo continua normalmente il proprio flusso di controllo.

La funzione pause() restituisce sempre -1 se l'errore errno = EINTR

Un'altra funzione simile a pause() e' sleep()

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds)
```

Tale funzione sospende il processo per un tempo pari all'argomento "seconds", il valore di ritorno e' 0 se la funzione e' terminata dopo il tempo stabilito, altrimenti restituisce il tempo "mancante".

La funzione sleep() puo' ritornare prima del previsto, in seguito alla ricezione di un segnale.

• Busy WAITING

Possiamo quindi attendere un segnale tramite pause() o sleep(), alternativamente durante l'attesa si possono eseguire altre istruzioni quest'ultimo metodo e' detto Busy WAITING.

TIMER

La LIBC mette a disposizione due funzioni che consentono di programmare l'invio di un segnale SIGALRM al processo stesso dopo un certo tempo, permettendo quindi di programmare un TIMER.

• ALARM

```
#include <unistd.h>
unsigned alarm(unsigned seconds)
```

- unsigned seconds: Indica l'intervalle di tempo dopo il quale il processo deve inviare a se' stesso un segnale SIGALRM, si osservi che il comportamento di default alla ricezione consiste nella terminazione del processo.

Se si effettua una successiva chiamata ad alarm() prima della ricezione del SIGALRM, la nuova chiamata reimposta un nuovo timer.

Se la seconda chiamata ha come argomento seconds=0, si elimina il timer impostato tramile la prima chiamata.

Tale funzione se risulta l'unica chiamata "attiva" restituisce 0, altrimenti restituisce il tempo mancante della precedente chiamata sovrascritta, NON risulta possibile il verificarsi di errori.

• VALARM ($u=\mu$ = microsecondi)

```
#include <unistd.h>
useconds_t valarm(useconds_t useconds, useconds_t interval)
```

- useconds.t useconds: Come alarm, ma timer impostato in microsecondi

- useconds.t interval: Dopo lo scadere del timer, reimposta un invio in loop di SIGALRM con cadenza pari a interval

Tale funzione gestisce la prima ricezione di SIGALRM dovuta alla terminazione del "primo" timer, ignorandola e continuando l'esecuzione, le successive ricezioni invece sono gestite in default (terminazione).

Il valore di ritorno e' analogo a quello di alarm()

INTERPRETAZIONI DICHIARAZIONI DI TIPO IN C

Vediamo ora un metodo algoritmico per comprendere facilmente il tipo di un dichiaratore: in uno dichiaratore C, naturalmente ha senso utilizzarla solo per espressioni complesse. Non immediate.

void * p(int) $p \rightarrow$ Dichiaraore

La procedura consiste nell'applicare i seguenti passi:

- PASSO 1: Partendo dal DICHIASTORE mi muovo a destra finche' non incontro punto e virgola o la parentesi tonda { ; ,) }
- PASSO 2: Partendo dal DICHIASTORE mi muovo a sinistra finche' posso oppure finche' incontro una parentesi tonda aperta "(" in tal caso il nuovo dichiaratore e' quelo racchiuso tra le tonde e toro al PASSO 1.

ESEMPIO 1

void * p(int)

p e' una funzione che ha come argomento un intero e che restituisce un puntatore di tipo void.

ESEMPIO 2

Void * (*(*p[5])())(int)

Void * (* DICHIASTORE ())(int)

Void * DICHIASTORE (int)

p e' un array di 5 puntatori / a funzione, senza parametri, che restituisce un puntatore/a funzione che ha come parametro un intero e che restituisce un puntatore a void

ALLOCAZIONE DINAMICA DELLA MEMORIA

Abbiamo visto precedentemente come mappare nuove pagine della memoria virtuale tramite `mmap()`, vediamo ora come allocare memoria nello heap. Al momento dell'avvio del processo gli unici blocchi della memoria logica che sono mappati sono i campi TEXT e parte di DATA, in particolare l'area riservata alle variabili globali NON INIZIALIZZATE che vengono imposte a zero, tale area è detta BSS. L'area di STACK e dello HEAP vengono invece mappate durante l'esecuzione del processo.

Inizialmente lo heap è vuoto, il puntatore che tiene traccia della zona allocata è BRK, per allocare memoria in tale spazio è sufficiente spostare BRK (meccanismo simile a %esp nel stack)



Per spostare tale puntatore sono disponibili due system call che non fanno più parte del standard POSIX ma che vediamo a scopo didattico e poi perché vengono ancora utilizzate nella LIBC per sistemi LINUX

```
#include <unistd.h>
int brk(void* addr)
void* sbrk(intptr_t increment).
```

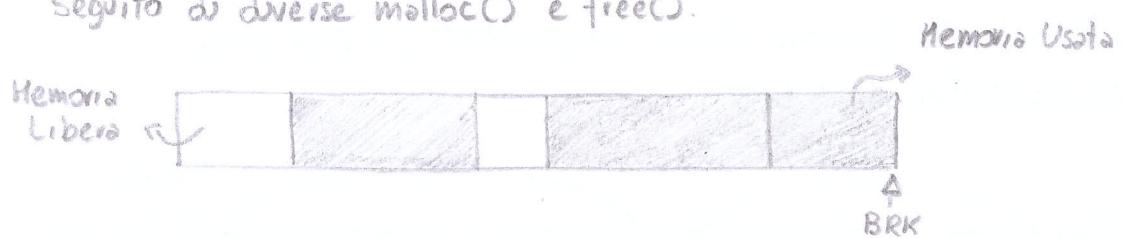
Il wrapper `brk()` impone `brk` ad un nuovo indirizzo puntato da `addr`. Se l'operazione va a buon fine restituisce zero, altrimenti -1. `sbrk()` invece incrementa il puntatore `brk` di una quantità positiva o negativa specificata dall'argomento, se ha successo restituisce il vecchio puntatore, altrimenti -1.

FUNZIONAMENTO ALLOCATORE

Considerando un architettura a 64 bit vediamo come funziona un semplice allocatore per lo heap, ovviamente non scendiamo in dettagli tecnici e ci concentriamo solo sull'idea di base.

Qualsiasi allocatore heap lavora sia in user mode che kernel mode, la modulazione kernel è necessaria per aumentare la dimensione dello heap nella memoria virtuale quando non ci sia spazio disponibile per un nuovo blocco nella memoria già mappata ma libera.

Partiamo dalla seguente osservazione, consideriamo il seguente heap a seguito di diverse malloc() e free().



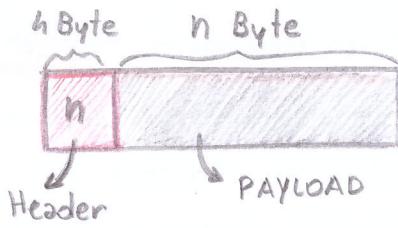
Occorre prima di tutto notare che tutto lo spazio del precedente heap è mappato in memoria, alcuni blocchi contengono dati, altri sono liberi, inoltre hanno dimensione VARIABILE non hanno una dimensione fissa come le pagine.

Dal precedente esempio notiamo che lo free() causa la comparsa di blocchi liberi di dimensione e posizione ignoti, tali informazioni vanno memorizzate e si fa aggiungendo METADATI ai singoli blocchi, tali dati si trovano solitamente all'inizio dei blocchi e sono detti appunto HEADER, la struttura dello heap diventa la seguente:



I blocchi occupati hanno la seguente struttura:

L'header ha una dimensione di 4 Byte e indica la grandezza del PAYLOAD, ovvero la parte del blocco che contiene il dato.



In questo modo abbiamo risolto il problema della memorizzazione della dimensione dei singoli blocchi, tuttavia non abbiamo modo di distinguere un blocco libero da uno occupato, si potrebbe pensare di aggiungere semplicemente un bit, ma tale metodo rende lenta la ricerca dei blocchi liberi, l'idea è quella di collegare i blocchi liberi tramite una lista utilizzando 8 Byte del payload come campo next. (8 Byte poiché siano in un architettura a 64 bit)

FREE_LIST



Si osservi che non si necessita di memorizzare anche i blocchi occupati dato che i puntatori a tali blocchi sono salvati dall'utente tramite appunto i puntatori, l'allocatore riottiene la posizione di un blocco quando viene invocata la `free()` la quale aggiunge semplicemente tale blocco in testa alla `free-list`.

OSS (Dimensione Minima Blocco)

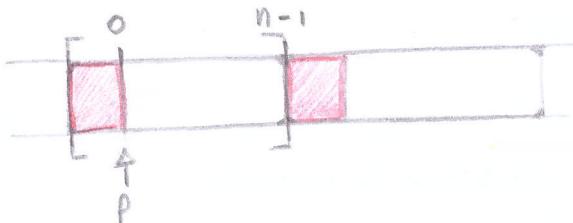
Osservando la struttura di un blocco libero, notiamo che ogni blocco deve avere una dimensione minima di $4+8=12$ Byte in modo da poter contenere l'header e il campo next per la `free-list`.

Al momento dell'allocazione con `malloc` quindi vengono automaticamente aggiunti 4 Byte per header e la dimensione minima del payload è sempre di 8 Byte (architettura 64 bit)

oSS (Buffer Overflow)

Consideriamo il seguente codice e heap:

```
char *p = malloc(n)
for(int i=0; i<=n; i++)
    p[i]=0;
```



Il precedente codice causa un buffer overflow che porta alla scrittura del byte più significativo dello header del blocco successivo, andando quindi a danneggiare la struttura.

Tale operazione NON porta ad un segmentation fault poiché tutta la memoria dello heap è allocata per il processo.

Per accorgersi di errori di questo tipo esistono tool appropriati, uno di questi è VALGRIND che vedremo successivamente

ESEMPIO DI SEMPLICE ALLOCATORE

Vediamo un esempio di funzionamento di un semplice allocator.

• ALLOC

Al momento dell'allocazione di un nuovo blocco possiamo trovarci in due casi differenti:

- CASO 1: Lo heap è vuoto, oppure non ci sono blocchi liberi abbastanza grandi.

In questo caso aumenta la dimensione tramite brk() o sbrk().

- CASO 2: Lo heap contiene blocchi liberi di dimensione adeguata.

In questo caso occorre scegliere uno dei diversi blocchi liberi che hanno la dimensione abbastanza grande per soddisfare la richiesta utente, per farlo vi sono diverse strategie.

- Strategie scelta blocchi liberi

Vi sono diverse strategie per la scelta di un blocco libero da restituire all'utente:

- Best-Fit: Scegliamo il blocco con dimensione minima ma sufficiente per soddisfare la richiesta.

Con tale tecnica minimizziamo lo spazio sprecato, ma occorre scorrere tutta la free.list.

- First-Fit: Scegliamo il primo blocco abbastanza grande.

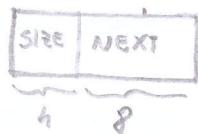
Tale tecnica rende il costo della ricerca nella free.list infatti solo nel caso peggiore si dovrà scorrere tutta la lista, tuttavia ha come svantaggio lo spreco di memoria

- Worst-Fit: Scegliamo il blocco più grande.

- Struttura blocchi liberi in C

In C possiamo usare la seguente struttura come header per i blocchi liberi, si osservi che è necessario introdurre la direttiva #pragma pack(1) in modo da evitare i 4 byte di padding causati dall'indirizzamento a 64 bit.

```
#pragma pack(1)
typedef struct header_t header_t
struct header_t {
    unsigned size;
    header_t *next;
}
```



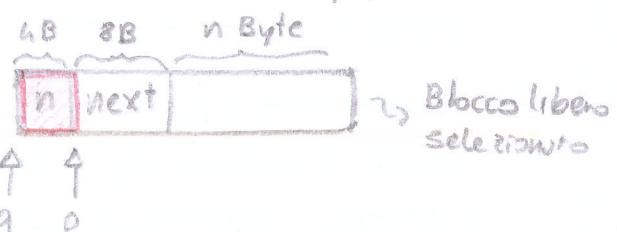
Nella malloc l'algoritmo ottiene tramite una funzione ausiliaria un puntatore ad un blocco libero, ma tale puntatore corrisponde all'header del blocco, all'utente invece dovremo restituire l'inizio del payload, occorre tenere presente l'aritmetica dei puntatori:

Vediamo un esempio di malloc:

```
char *malloc(size_t n)
    header_t *q = getFreeBlock(n); // Restituisce un blocco con payload n
    char* p = (char*)q + h;
    return p;
```

Il cast è necessario altrimenti per l'aritmetica dei puntatori in C, poiché q è di tipo header_t si avrebbe:

$$p = q + \text{sizeof(header_t)}$$



FREE

Al momento della free() invece l'algoritmo deve aggiungere il blocco alla free-list, ma l'utente fornisce un puntatore al payload e non al header.

```
void free(void* p)
    header_t *q = (char*)p - h;
    add-free-list(q); // mette q in testa alla free-list
```

Perché è necessario il cast a char*?

Che dimensione ha void*, forse zero?

• OTIMIZZAZIONI

Possiamo fare diverse ottimizzazioni per il nostro allocatore:

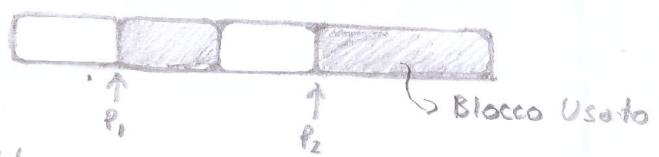
1. UNIONE BLOCCHI LIBERI ADIACENTI: Al momento della free() se due o più blocchi adiacenti sono liberi li unisco in un unico blocco.
2. DIVISIONE BLOCCHI: Al momento della malloc, se si ha un blocco libero di dimensione molto più grande di quella richiesta, possiamo dividere il blocco libero in modo da non sprecare memoria.
3. ALLINEAMENTO INDIRIZZI DEI BLOCCHI: Allineando gli indirizzi dei blocchi a multipli di 8 Byte o 16 Byte, si velocizza l'accesso a memoria, nonostante lo spreco di memoria e' sempre conveniente guadagnare in prestazioni.
4. DIVERSE FREE-LIST: Usando blocchi di diverse dimensioni standard ad esempio 16B - 24B - 32B, possiamo velocizzare la ricerca dei blocchi liberi, montando diverse free-list, ognuna avente tutti i blocchi con la medesima dimensione.

OSS (Frammentazione Esterna)

Utilizzare blocchi di dimensione variabile causa la frammentazione esterna, ovvero nonostante si abbia memoria libera sufficiente per uno malloc(), non si dispone di memoria contigua.

Si potrebbe pensare di effettuare una deframmentazione come nei filesystem

Windows, ma ciò non è possibile poiché il blocco puntato da P_1 non può essere spostato, dato che renderebbe non più valido l'indirizzo precedentemente fornito all'utente.



OSS (Scelta Blocco) (Best-Fit)

Si osservi che la tecnica di Best-Fit per la scelta dei blocchi non è sempre la migliore poiché usando l'ottimizzazione di divisione dei blocchi, si creano blocchi troppo piccoli che raramente vengono utilizzati.



FRAHMMENTAZIONE DELLA MEMORIA

Per FRAHMMENTAZIONE DELLA MEMORIA si intende spazio allocato, ovvero mappato nella memoria fisica ma non utilizzabile, ovvero memoria sprecata. Il problema della frammentazione colpisce qualsiasi tipo di allocator, sia per la memoria centrale (RAM) che per il disco rigido nel caso dei filesystem.

Vi sono due tipi di frammentazione:

- INTERNA: Spreco nei blocchi
- ESTERNA: Spreco tra i blocchi.

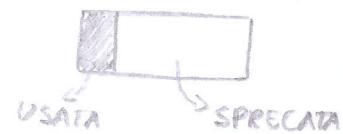
• FRAHMMENTAZIONE INTERNA

Per frammentazione INTERNA si intende spreco di memoria all'interno dei blocchi allocati.

Vediamo alcuni esempi:

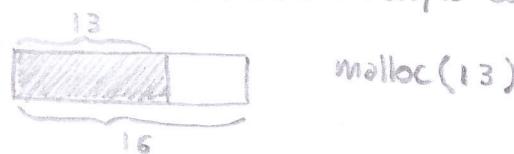
- Memoria Virtuale

La memoria virtuale dato che è suddivisa in blocchi di dimensione fissa (PAGINE) soffre di frammentazione interna dato che l'utente potrebbe necessitare di uno spazio di memoria molto più piccolo di una pagina.



- Allocator Malloc

Se l'allocator malloc utilizza blocchi con dimensione multiplo di 4 Byte o 8 Byte, soffre di frammentazione interna.



- Padding nelle Struct

Anche le struct in C soffrono di frammentazione interna a causa dell'allineamento degli indirizzi dei campi.

```
struct {
    char v;
    char* p;
    short s;
}
```



OSS (Ridurre Frammentazione Interna)

Per ridurre la frammentazione interna occorre scegliere le dimensioni dei blocchi più piccola possibile, ad esempio le pagine nella memoria virtuale sono di 4096 Byte

• FRAHMMENTAZIONE ESTERNA

Per frammentazione ESTERNA si intende spreco di memoria al di fuori dei blocchi, ovvero tra un blocco e l'altro.

Un esempio di allocatore che soffre di frammentazione esterna è quello dello heap ovvero la malloc:



OSS (Allocatore Memoria Virtuale)

Si osservi che l'allocatore della memoria virtuale non soffre di frammentazione esterna dato che i blocchi hanno una dimensione fissa, quindi se si ha a disposizione un blocco libero (pagina) si potrà sempre soddisfare la richiesta

VALGRIND

Valgrind è una famiglia di tool utili per il debugging e la profilazione di programmi, noi utilizzeremo il tool MEMCHECK il quale consente di individuare errori sulla gestione della memoria allocata nello heap, ad esempio memory leak oppure accesso a blocchi liberati con free().

Questi tipi di errori non possono essere individuati a tempo di compilazione e potrebbero non esserlo nemmeno durante l'esecuzione dei test, strumenti come Valgrind risultano quindi fondamentali.

La sintassi con cui lanciate Valgrind è la seguente:

valgrind [valgrind-options] [your-program] [your-program-options]

Vi sono diverse option per valgrind, la più usata è --tool che consente di selezionare lo specifico tool, nel nostro caso:

--tool=memcheck

Il tool memcheck è selezionato di default, non è quindi necessario specificarlo con --tool

oss (--)

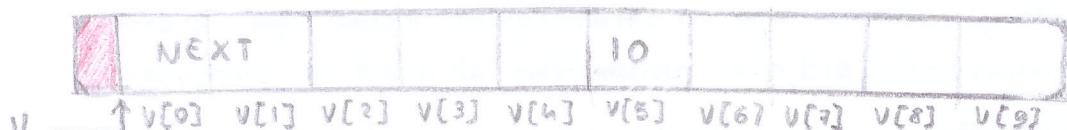
I doppi trattini servono per indicare che tool è una parola e non l'unione di più argomenti

Il programma profilato deve essere compilato con lo specifico di debug -g (come per gdb)

ESEMPIO (et-access-freeed)

Consideriamo il seguente programma che accede ad una area di memoria dopo averla liberato con free().

La printf fa un accesso a memoria NON corretto, tuttavia NON causa un segmentation fault dato che la zona in cui accede è ancora allocata al processo, inoltre il programma stampa correttamente 10 per il seguente motivo:



Una volta liberato il blocco puntato da v, l'allocatore aggiunge il blocco alla free-list usando i primi 8 byte come campo next (Arch 64bit), gli altri byte sono lasciati invariati.

Si osservi che alcuni allocator potrebbero usare una posizione diversa per il campo next.

```

1 ==5866== Memcheck, a memory error detector
2 ==5866== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
3 ==5866== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
4 ==5866== Command: ./e1-access-freed → Comando con cui e' stato lanciato il programma
5 ==5866=
6 ==5866== Invalid read of size 4 → Unico errore
7 ==5866== at 0x108712: main (e1-access-freed.c:8)
8 ==5866== Address 0x522d054 is 20 bytes inside a block of size 40 free'd
9 ==5866== at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
10 ==5866== by 0x108709: main (e1-access-freed.c:7)
11 ==5866== Block was alloc'd at
12 ==5866== at 0x4C2FB0F: malloc (in
13 /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
14 ==5866== by 0x1086EB: main (e1-access-freed.c:5)
15 ==5866= 10 → OUTPUT programma
16 ==5866=
17 ==5866== HEAP SUMMARY:
18 ==5866==   in use at exit: 0 bytes in 0 blocks
19 ==5866==   total heap usage: 2 allocs, 2 frees, 4,136 bytes allocated
20 ==5866=
21 ==5866== All heap blocks were freed -- no leaks are possible
22 ==5866=
23 ==5866== For counts of detected and suppressed errors, rerun with: -v
24 ==5866== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
25

```

↳ PID Valgrind

Compilando il programma e lanciando Valgrind tramite i seguenti comandi otteniamo il precedente output.

>> gcc -O e1-access-freed e1-access-freed -g
 >> valgrind ./e1-access-freed

Analizzando l'output notiamo che tutte righe, eccetto la n°15, iniziano con un numero che corrisponde al PID di Valgrind. La riga 15 corrisponde invece all'output del nostro programma, infatti Valgrind durante l'analisi esegue il nostro codice.

L'output e' diviso in diverse parti: (successivamente ne vedremo un'altra)

1. INTRO: Informazioni riguardanti Valgrind

2. COMANDO: Comando con cui e' stato lanciato il programma

3. ESECUZIONE: In questa parte Valgrind esegue il nostro programma riportando gli eventuali errori, ci saranno quindi alcune righe, le quali non iniziano con il pid, che corrispondono all'output del programma

4. HEAP SUMMARY: Riepilogo sull'uso della gestione della memoria nello heap.

5. ERROR SUMMARY: Riepilogo errori.

(6. MEMORY LEAK)

Osservando il blocco 3 notiamo che c'è presente solo un errore "Invalid read of size 4", le righe successive infatti sono identificate internamente. Tale errore si riferisce, come indicato, alla riga 8 del main, corrispondente alla printf, Valgrind inoltre ci segnala lo spostamento ($4 \times 5 = 20$ Byte) nel blocco dello heap in cui leggeremo erroneamente i 4 Byte poiché appartenenti ad un blocco liberato mediante la free() del main (riga 9-10), oss (free-malloc di Valgrind)

Nelle righe 9 e 12 notiamo che Valgrind utilizza funzioni free() e malloc() diverse da quelle della LIBC, le quali consentono il tracciamento della gestione della memoria

OSS (Heap Summary)

Nello heap summary, Valgrind segnala l'allocazione e la liberazione di due blocchi, il nostro main alloca un solo blocco, tuttavia la printf ne alloca un altro corrispondente al buffer di output

ESEMPIO (e1-ujmp.c) (Salto diperde da variabile non inizializzata)

Un altro errore che non può essere notato a tempo di compilazione è il passaggio ad un funzione di un parametro corrispondente ad una variabile NON INIZIALIZZATA; anche questo tipo di errore viene segnalato da Valgrind.

Un esempio di errore di questo tipo è il seguente:

```
| #include <stdio.h>
| void foo(int *a)
|   if(a==NULL)
|     return;
|   printf("a is %d/n", *a);
|
| int main()
|   int *a; // VARIABILE NON INIZIALIZZATA
|   foo(a);
|   return 0;
```

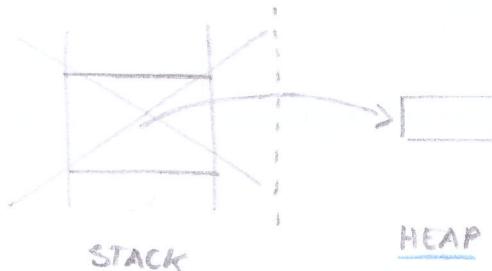
ESEMPIO (el-memleak) (memory leak)

```
7 #include <stdlib.h>
8 int main()
9     int* v = malloc(10*sizeof(int));
10    return 0;
11
12 ==15537== Memcheck, a memory error detector
13 ==15537== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
14 ==15537== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
15 ==15537== Command: ./el-memleak
16 ==15537==
17 ==15537==
18 ==15537== HEAP SUMMARY:
19 ==15537==     in use at exit: 40 bytes in 1 blocks
20 ==15537==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
21 ==15537==
22 ==15537== LEAK SUMMARY:
23 ==15537==   definitely lost: 40 bytes in 1 blocks
24 ==15537==   indirectly lost: 0 bytes in 0 blocks
25 ==15537==   possibly lost: 0 bytes in 0 blocks
26 ==15537==   still reachable: 0 bytes in 0 blocks
27 ==15537==   suppressed: 0 bytes in 0 blocks
28 ==15537== Rerun with --leak-check=full to see details of leaked memory
29 ==15537==
30 ==15537== For counts of detected and suppressed errors, rerun with: -v
31 ==15537== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
32
```

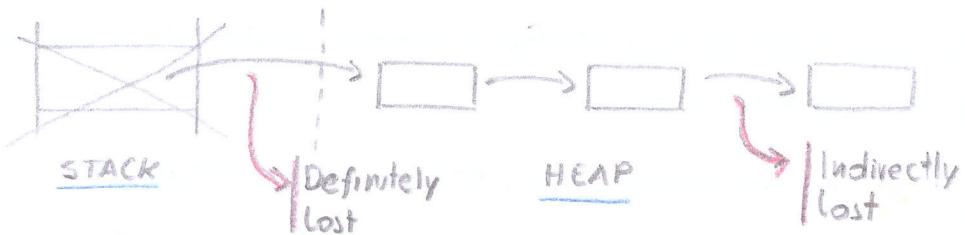
In questo caso notiamo che l'errore è di tipo memory leak, ovvero durante l'esecuzione perdiamo il puntatore restituito dallo malloc. Senza aver prima deallocato il blocco corrispondente. Nel nostro esempio perdiamo il puntatore al momento della terminazione del main, ovvero quando viene eliminato il record di attivazione, contenente v, perdiamo il puntatore al blocca.

Nella sezione LEAK SUMMARY osserviamo i vari tipi di memory leak che possono avvenire:

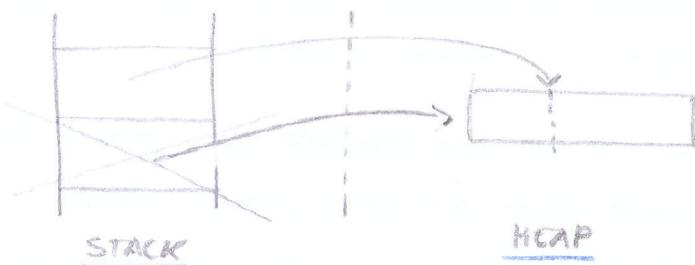
- DEFINITELY LOST: Corrisponde al caso tipico in cui si perde il puntatore del blocco restituito da malloc



- INDIRECTLY LOST: Corrisponde alla perdita indiretta di un puntatore causata dal definitely lost di puntatore precedente.
Un errore di questo tipo puo' avvenire nel caso di una lista collegata:



- Possibly lost: Corrisponde a casi particolari per cui si perde il puntatore esplicito all'inizio del blocco, ma si olspre tuttavia da un altro che punta ad un'altra parte del blocco stesso, quando successivamente e' ancora possibile deallocarlo. Valgrind segnala comunque l'errore poiche' potrebbe essere invobitario.



- STILL REACHABLE: Corrisponde a variabili statiche o globali allocate tramite malloc, infatti tali variabili devono durare l'intera esecuzione del processo e quindi si potrebbe evitare di dealloarle.

N.B. (GESTIONE MEMORIA STACK)

Si noti che Valgrind Non e' in grado di rilevare errori sulla gestione della memoria sullo stack, ad esempio il seguente errore di lettura e scrittura di un array non viene rilevato.

```
int main()
{
    int buf[3] = {1, 2, 3};
    buf[4] = 10;
    printf("%d\n", buf[4]);
    return 0;
```

VARIABILI DI AMBIENTE

Le VARIABILI DI AMBIENTE sono variabili non associate ad singolo processo, ma condivise all'interno del sistema operativo (ambiente).
Tali variabili sono tutte di tipo stringa e vengono modificate o dichiarate da terminale nel seguente modo:

>> Nome=Valore (es: x=5, x e' una stringa '5')

Si osservi che la precedente assegnazione di valore NON deve contenere alcuno spazio, inoltre introduce una nuova variabile o cambia il suo valore precedente solo nel terminale attuale.

Tramite il comando "env" possiamo vedere tutte le variabili di ambiente presenti nel relativo terminale (>> env).

Per poter esportare una variabile di ambiente, visibile solo ai figli del relativo terminale si utilizza il comando "export":

>> export PIPPO=10 // Variabile condivisa con i processi figli

>> PIPPO=10 // Variabile valida solo per il processo attuale

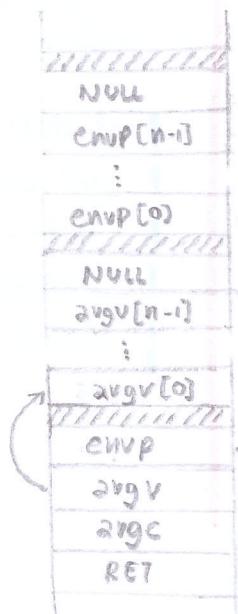
Inoltre c'è possibile aggiungere tale comando nel file ".bashrc" in modo che venga eseguito all'avvio del primo terminale e quindi sarà disponibile in tutti i terminali che verranno lanciati successivamente.

VARIABILI DI AMBIENTE IN C

Fino adesso abbiamo visto come gestire le variabili di ambiente da terminale (bash). Vediamo ora come farlo in un programma C. Aggiungendo un argomento al main possiamo ottenere una copia delle variabili di ambiente nel nostro stack:

```
int main(int argc, char *argv[], char *envp[])
{
    char **e = envp;
    while(*e) printf("%s\n", *e++);
}
```

L'output del precedente programma è esattamente uguale a quello ottenuto con >> env.



Per la gestione delle variabili di ambiente sono disponibili anche le seguenti funzioni `setenv()` e `getenv()`.

- `Setenv()`

`int setenv(const char* name, const char* value, int overwrite)`

Tramite `setenv()` possiamo dichiarare una nuova variabile di ambiente oppure reimpostare una già esistente se l'argomento `overwrite` è settato ad 1.

Tale funzione NON cambia le variabili di ambiente ottenute dal main tramite il passaggio per argomenti dato che esse sono memorizzate nel stack, tuttavia consente di esportare gli aggiornamenti ai processi figli.

- `Getenv()`

`char* getenv(const char* name)`

Tale funzione restituisce un puntatore contenente il valore (stringa) della variabile di ambiente "name", se questo non esiste ritorna NULL

- `environ`

In C, invece di ottenere le variabili di ambiente tramite stack, si può utilizzare una variabile globale estesa della LIBC contenente un puntatore all'array di variabili di ambiente, tale variabile è "environ" ed è definita nel file `unistd.h`.

Vediamo un esempio:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
extern char** environ // Definito in unistd.h
int main(int argc, char* argv[])
{
    char** e = environ;
    while(*e) printf("%s\n", *e++);
    return 0;
}
```

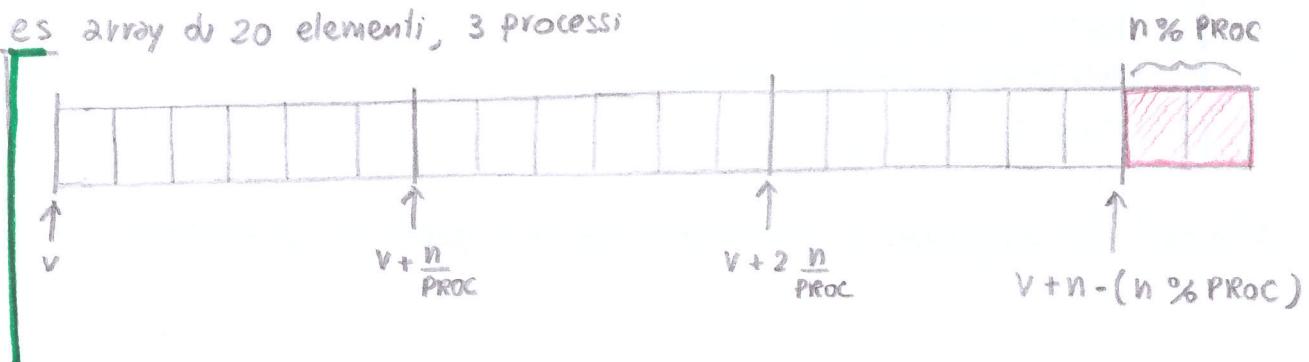
L'output del programma è esattamente uguale a quello del precedente.

ESERCIZIO (Ricerca Parallelta)

```
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <unistd.h>
10 #include <sys/types.h>
11 #include <sys/wait.h>
12 #include "../e4.h"
13
14 #define PROC 2
15 int find(int* v, unsigned n, int x) {
16     int i;
17     for (i=0; i<n; ++i)
18         if (v[i]==x) return 1;
19     return 0;
20 }
21 int par_find(int* v, unsigned n, int x){
22     int i, res = 0;
23     for (i=0; i<PROC; ++i) {
24         pid_t pid = fork();
25         if (pid == -1) {
26             perror("fork");
27             _exit(EXIT_FAILURE);
28         }
29         if (pid == 0) {
30             int res = find(v+i*n/PROC, n/PROC, x);
31             _exit(res);
32         }
33     }
34     for (i=0; i<PROC; ++i) {
35         int status;
36         wait(&status);
37         if (WIFEXITED(status)) {
38             res = res || WEXITSTATUS(status);
39         }
40     }
41     return res || v[n-1] == x; // Funziona solo per resto 0 o 1.
42 }
43 } return res || find(v+n-(n%PROC), n%PROC, x);
```

Il precedente programma divide l'array nel seguente modo:

es array di 20 elementi, 3 processi



ESERCIZIO (Confronto file)

```
7 #include "../e5.h"
8 #include <errno.h>
9 #include <stdlib.h>
10 #include <sys/types.h>
11 #include <sys/stat.h>
12 #include <fcntl.h>
13
14 #define BUF_SIZE 4096
15 int file_eq(char* f1, char* f2) {
16     int res, equal = 1;
17     char* buf1 = NULL;
18     char* buf2 = NULL;
19     int fd1 = -1;
20     int fd2 = -1;
21     buf1 = malloc(BUF_SIZE);
22     if (buf1 == NULL) goto error;
23     buf2 = malloc(BUF_SIZE);
24     if (buf2 == NULL) goto error;
25     fd1 = open(f1, O_RDONLY);
26     if (fd1 == -1) goto error;
27     fd2 = open(f2, O_RDONLY);
28     if (fd2 == -1) goto error;
29     for (;;) {
30         int r1 = read(fd1, buf1, BUF_SIZE);
31         if (r1 == -1) goto error;
32         int r2 = read(fd2, buf2, BUF_SIZE);
33         if (r2 == -1) goto error;
34         if (r1 != r2 || memcmp(buf1, buf2, r1) != 0) {
35             equal = 0;
36             break;
37         }
38         if (r1 == 0) break;
39     }
40     free(buf1);
41     free(buf2);
42     res = close(fd1);
43     if (res == -1) goto error;
44     res = close(fd2);
45     if (res == -1) goto error;
46     return !equal;
47 error: ;
48     int e = errno;
49     if (buf1 != NULL) free(buf1);
50     if (buf2 != NULL) free(buf2);
51     if (fd1 != -1) close(fd1);
52     if (fd2 != -1) close(fd2);
53     errno = e;
54     return -1;
55 }
56 }
```

memcmp è simile a strcmp, ma al contrario di quest'ultima lavora con buffer generici senza necessariamente il terminatore di stringhe '\0'

Se i buffer sono uguali, memcmp() restituisce 0

Per generare file casuali :

>> dd if=/dev/random of=file.txt count=8192 bs=4096

Numeri
Blocchi

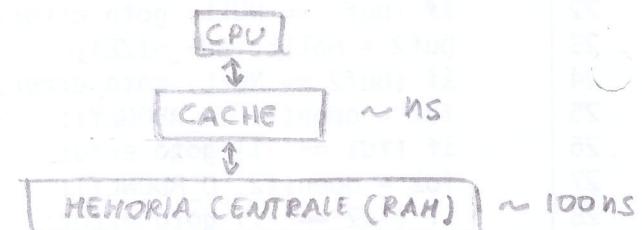
Dimensione
Blocchi

MEMORIA CACHE

L'accesso a memoria è una delle operazioni più costose a livello temporale, infatti l'accesso alla RAM ha un costo circa 100 volte maggiore rispetto alla lettura di un registro. Per ridurre tale gap, tra la CPU e la RAM è presente un'altra memoria molto più veloce chiamata MEMORIA CACHE.

La memoria cache è costruita con tecnologia SRAM (static RAM), che consente accessi a memoria molto più veloci, tuttavia ha un costo per byte molto più elevato rispetto alla RAM costruita con tecnologia DRAM (Dynamic RAM). Per questo motivo la memoria cache è molto più piccola rispetto alla RAM. Se così non fosse il costo di un computer sarebbe esorbitante.

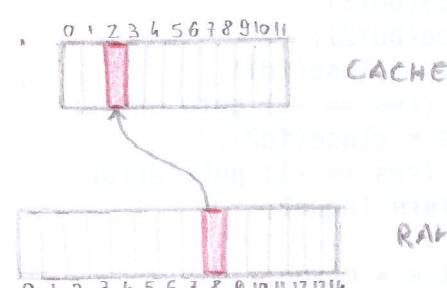
Lo scopo della cache è quello di tenere una COPIA dei dati più frequentemente acceduti dalla CPU senza dover ogni volta accedere alla RAM.



DIVISIONE LOGICA CACHE

Dato che la cache ha lo scopo di contenere una copia di alcuni dati memorizzati in RAM è necessario definire cosa si intende per dato e qual è l'unità di base. In modo simile alla suddivisione della memoria fisica in frame e di quella virtuale in pagine, la cache e la memoria centrale (fisica) sono divise in unità di base della stessa dimensione dette BLOCCHI.

Quando un processo accede ad un indirizzo x il gestore della memoria deve calcolare a quale blocco appartiene tale indirizzo e per farlo esegue una divisione intera di x per la dimensione B dei blocchi.



x : Indirizzo logico

B : Dimensione in byte
dei blocchi

$$\text{Blocco}(x) = \lfloor \frac{x}{B} \rfloor$$

Soltamente i blocchi della cache hanno dimensione di 64 Byte, sono quindi molto più piccoli delle pagine ($\approx 4\text{ kB}$)

CACHE HIT / MISS

Una volta calcolato il blocco corrispondente ad un determinato indirizzo logico x possono avvenire due casi:

- CACHE HIT: Il blocco richiesto e' già nella memoria cache quando la CPU vi accede direttamente
- CACHE MISS: Il blocco richiesto NON e' presente nella cache, si procede quindi a copiarlo dalla RAM alla CACHE e poi effettuare un accesso in quest'ultima

Si osservi che la CPU non accede mai direttamente alla RAM, prima di leggere un dato, in caso di cache miss, lo porta prima di tutto in cache e poi effettua la lettura

In caso di cache miss se sono presenti blocchi liberi allora si procede semplicemente a caricare il blocco desiderato, altrimenti occorre scegliere un blocco "vittima" da eliminare. Se il blocco vittima e' stato modificato allora occorre prima di tutto aggiornare il suo valore nella RAM e poi puo' essere eliminato. Deduciamo quindi che i cache miss sono da evitare poiche' richiedono un considerevole dispendio di tempo, occorre quindi scegliere una politica di rimpiazzo dei blocchi adeguata, ovvero come selezionare un blocco vittima. La scelta piu' usata e' quella di scegliere il blocco usato meno recentemente LRU (Least Recently Used) poiche' si presume che tale blocco non verrà usato nell'immediato futuro.

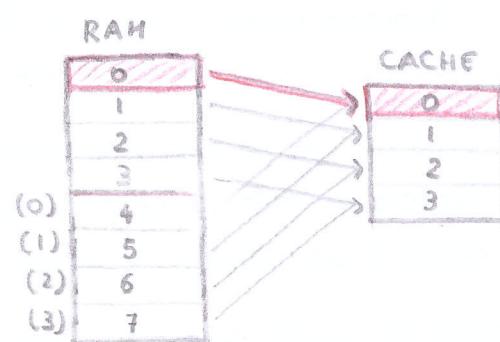
ASSEGNAZIONE BLOCCHI (Associatività)

Quando la CPU deve accedere ad un indirizzo x il gestore della memoria calcola il blocco corrispondente e lo cerca nella cache, se l'assegnazione tra blocco della RAM e della CACHE fosse casuale la ricerca impiegherebbe troppo tempo inoltre andrebbe ad aggravare ulteriormente il cache miss.

Le principali tecniche di assegnazione sono le seguenti:

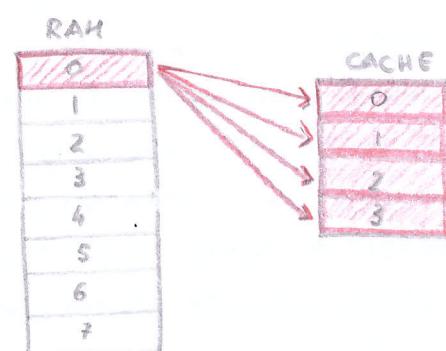
- Cache ad Accesso Diretto

In questa tecnica, la più restrittiva, si suddivide la RAM in sezioni pari alla dimensione della cache e ogni blocco di tali sezioni può essere associato solo alla relativa riga, non possiamo quindi applicare la politica LRU.



- Cache Completamente Associativa

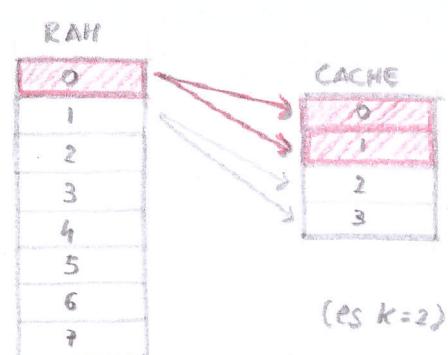
Tale tecnica è esattamente l'estremo opposto della precedente, in questo caso infatti un blocco può essere caricato in qualsiasi linea. Possiamo quindi usare LRU, ma la ricerca dei blocchi nella cache è molto più lenta.



- Cache Associativa a K vie

La tecnica più utilizzata è l'associatività a K vie, ovvero un blocco può essere associato scegliendo K diverse linee (vie). Si osservi che tale tecnica è una generalizzazione delle precedenti, per $K=1$ si ha quella ad accesso diretto e per $K = \text{dim}(\text{cache})$ si ha quella completamente associativa.

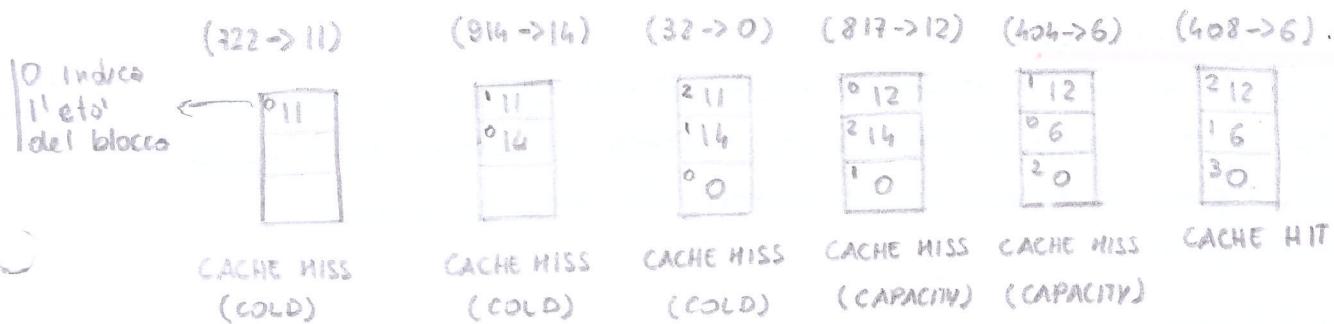
Anche in questa tecnica possiamo applicare LRU.



- Esempio (Cache Completamente Associativa)

Vediamo un esempio dell'evoluzione della memoria cache di tipo COMPLETAMENTE ASSOCIATIVA con 3 blocchi da 6h Byte, considerando l'accesso ai seguenti indirizzi $\{722, 914, 32, 817, 404, 408\}$

Primo di tutto calcoliamo i blocchi corrispondenti ai precedenti indirizzi, per farlo e' sufficiente dividerli per la dimensione dei blocchi, nel nostro caso 6h Byte.



- Tipi di Cache Miss

I CACHE MISS possono essere classificati in tre categorie:

- COLD: Si ha quando il caricamento del blocco non ha causato l'eliminazione di un altro blocco. (Detto anche COMPULSORY)

- CAPACITY: Si ha quando l'intera cache e' piena e quindi e' necessario l'eliminazione di un blocco

- CONFLICT: Si ha quando per caricare un blocco e' necessario sostituire un blocco esistente a causa dei vincoli di associatività nonostante ci siano linee libere.

PRINCIPIO DI LOCALITÀ

Il precedente meccanismo basato sull'uso di una memoria cache come intermediario tra la CPU e la DRAM, non avrebbe senso se si accedesse ad indirizzi casuali poiché si avrebbe un cache miss ad ogni accesso e ciò peggiorerebbe notevolmente le prestazioni. Fortunatamente per come viene eseguito un programma gli accessi non sono casuali ma tendono ad esibire due proprietà dette PRINCIPIO DI LOCALITÀ:

• LOCALITÀ TEMPORALE

La proprietà di LOCALITÀ TEMPORALE consiste nel fatto che se un programma accede ad un indirizzo è probabile che vi accederà nuovamente nell'immediato futuro. Si pensi ad esempio ad una funzione che lavora con le proprie variabili locali.

• LOCALITÀ SPAZIALE

La proprietà di LOCALITÀ SPAZIALE consiste nel fatto che se un programma accede ad un indirizzo è probabile che nell'immediato futuro accederà ad indirizzi vicini. Si pensi ad esempio alla scansione sequenziale di un array.

Soltanente tali proprietà emergono spontaneamente, tuttavia a volte è necessaria l'accortezza del programmatore.

• ESEMPIO (Accesso Matrici)

Un esempio di operazione in cui si può far emergere la località spaziale è l'accesso ad una matrice, infatti se si accedono gli elementi per riga si avranno un numero nettamente minore di cache miss rispetto all'accesso per colonne, poiché gli elementi sono "vicini" e quindi saranno nel medesimo blocco il quale rimarrà più tempo in cache.

Consideriamo una matrice di interi 2×8 ed una cache con una linea di 16 Byte

1 ^o Riga				2 ^o Riga			
1	2	3	4	5	6	7	8

(Per Righe) (2 cache miss)

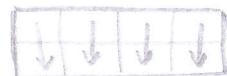
In questo caso accedendo al primo elemento carico un blocco contenente l'intera riga, i successi accessi saranno quindi cache hit



1 ^o Riga				2 ^o Riga			
1	3	5	7	2	4	6	8

(Per colonne) (8 cache miss)

In questo caso ad ogni accesso si deve caricare il blocco corrispondente



Esempio (Moltiplicazione Matrici)

```
7 CC = gcc
8 CFLAGS = -O1
9
10 all: e2_main.c
11     $(CC) $(CFLAGS) -DVERSION=0 e2_main.c -o e2-ijk
12     $(CC) $(CFLAGS) -DVERSION=1 e2_main.c -o e2-ikj e2-kij
13     $(CC) $(CFLAGS) -DVERSION=2 e2_main.c -o e2-jki
14
15 .phony: clean
16     Dichiara che clean non e' un target di compilazione
17 clean:
18     rm -rf e2-ijk e2-ikj e2-kij
19
20 #include <stdio.h>
21 #include <stdlib.h>
22 #include <string.h>
23 #define N 1024
24 double A[N][N];
25 double B[N][N];
26 double C[N][N];
27
28 int main() {
29     int i, j, k;
30     printf("Initializing matrices...\n");
31     // initialize A
32     for (i = 0; i < N; i++) {
33         for (j = 0; j < N; j++) {
34             A[i][j] = i*j;
35         }
36     }
37     memcpy(B, A, sizeof(A));
38     memset(C, 0, sizeof(C));
39     printf("Multiplying matrices...\n");
40     #if VERSION == 0 // ijk scheme → time = 16.468s
41         for (i=0; i<N; i++)
42             for (j=0; j<N; j++)
43                 for (k=0; k<N; k++)
44                     C[i][j] += A[i][k] * B[k][j];
45     #elif VERSION == 1 // kij scheme → time = 2.086s (MIGLIORE)
46         for (k=0; k<N; k++)
47             for (i=0; i<N; i++)
48                 for (j=0; j<N; j++)
49                     C[i][j] += A[i][k] * B[k][j];
50     #elif VERSION == 2 // jki scheme → time = 37.412s
51         for (j=0; j<N; j++)
52             for (k=0; k<N; k++)
53                 for (i=0; i<N; i++)
54                     C[i][j] += A[i][k] * B[k][j];
55     #endif
56     return 0;
57 }
```

Dipendenze del target all

#define VERSION 0

Dichiara che clean non e' un target di compilazione

Memcpy(B, A, sizeof(A));

printf("Multiplying matrices...\n");

#if VERSION == 0 // ijk scheme → time = 16.468s

#elif VERSION == 1 // kij scheme → time = 2.086s (MIGLIORE)

#elif VERSION == 2 // jki scheme → time = 37.412s

Mi sposto su C e B per righe, A[][] e' costante

(PEGGIORI) Poiche' mi sposto su C ed A per colonna.

GERARCHIE DI MEMORIA

Nel sistema moderno sono presenti tre livelli di cache, ognuno dei quali si comporta come descritto precedentemente. La CPU accede direttamente solo al livello più alto L1, ovvero quello più veloce.

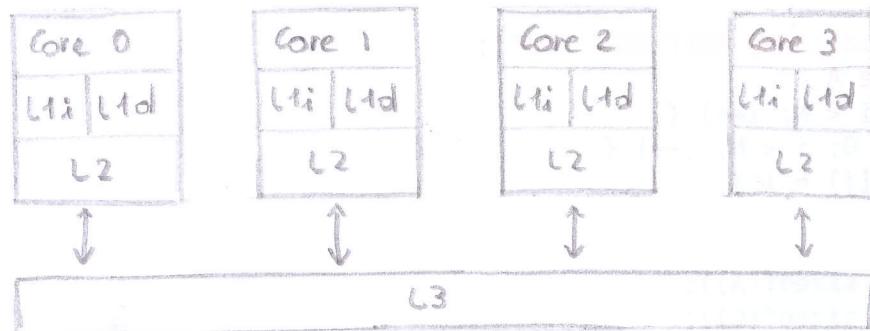
• CACHE L1

Nelle CPU moderne si hanno due cache L1:

- L1i: Cache L1 usata solo per il caricamento delle istruzioni in eip
- L1d: Cache L1 usata per caricare i normali dati

• SISTEMI MULTICORE (Intel Core i7)

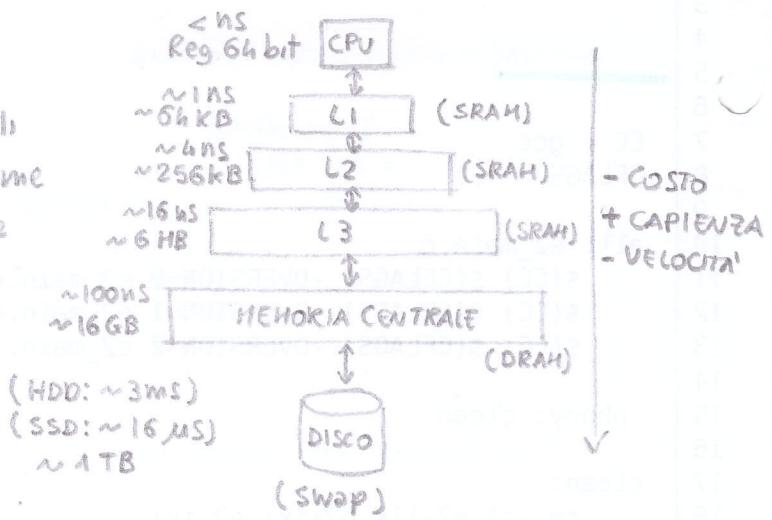
Nel sistema multicore moderno, ad esempio un processore Intel Core i7, si hanno più CPU le quali abbiano ognuna la propria L1 e L2, condividendo la L3.



TIPI DI CACHING

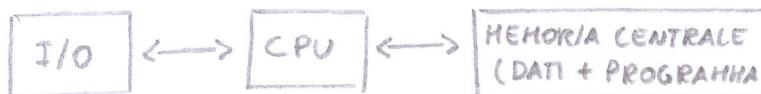
Il meccanismo di caching visto per la memoria si applica in diversi contesti:

- CACHE - CPU: Quello appena visto
- SWAPPING PAGINE: Spostamento pagine nello swap (disco)
- CACHING DISCO: Durante la lettura di un file, il sistema operativo carica in RAM interi blocchi, non solo i byte richiesti.
- CACHING PAGINE WEB: Il web browser memorizza localmente alcune componenti di una pagina web in modo da velocizzare il caricamento successivamente
- WEB PROXY: Per ridurre la latenza tra il nostro PC e un server, vi si interpone un PROXY più vicino geograficamente, il quale memorizza le pagine accedute, in modo da velocizzare una successiva richiesta

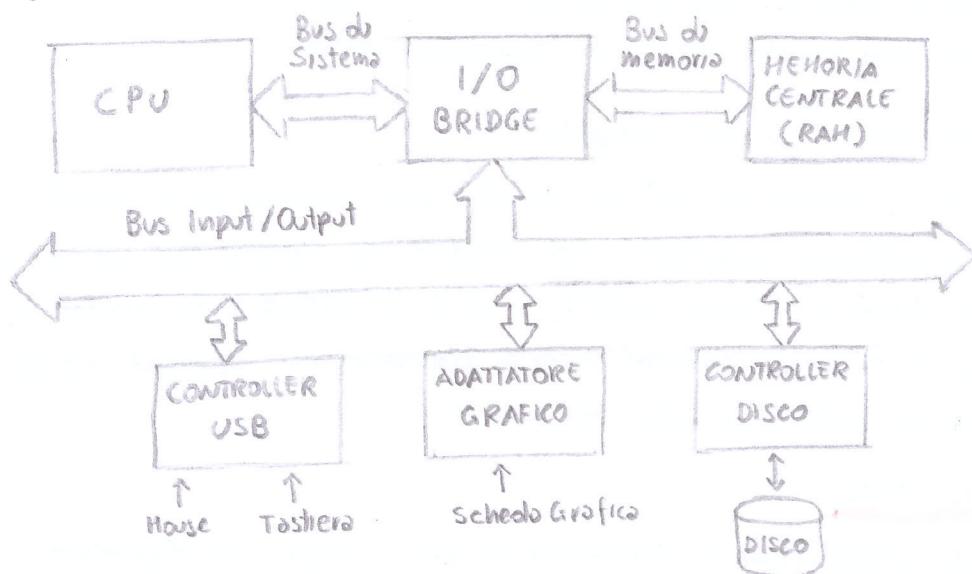


ARCHITETTURA DI UN CALCOLATORE

Un calcolatore puo' essere rappresentato mediante il modello di Von Neumann:



Tuttavia piu' in dettaglio, l'architettura di un calcolatore moderno e' la seguente:



La CPU comunica con le altre componenti mediante I/O Bridge e i relativi bus, inoltre per comunicare con i dispositivi di input / output necessita di controller (incorporato nella scheda madre) oppure di un adattatore (schede esterne collegate alla scheda madre)

Nella CPU sono presenti numerosi componenti, i piu' importanti sono:

- REGISTER FILE: Registri
- ALU (Arithmetic Logic Unit): Esegue operazioni aritmetico-logiche
- FPU (Floating Point Unit): Esegue operazioni aritmetiche su dati floating point
- AGU (Address Generation Unit): Calcola gli indirizzi (LEA)
- UNITÀ DI CONTROLLO: Esegue un microcodice il quale scomponete in varie fasi l'esecuzione di una singola istruzione

TIPI DI ARCHITETTURE

Le architetture utilizzate nelle CPU moderne possono essere classificate in varie diverse tipologie:

• RISC (Reduce Instruction Set Computer)

I processori RISC hanno un set di istruzioni ridotto con l'obiettivo di utilizzare un hardware più semplice e quindi che consumi meno energia.

Tale tipologia è adottata nelle architetture MIPS, PowerPC, SPARC e ARM, quest'ultima molto diffusa nei dispositivi mobili.

• CISC (Complex Instruction Set Computer)

I processori CISC hanno un set di istruzioni più complesso e quindi un hardware altrettanto complesso, ciò consente di svolgere una determinata operazione con meno istruzioni rispetto a RISC. Un esempio di architettura CISC è X86, in cui come abbiano visto è possibile avere operandi memoria al contrario di un architettura RISC come ARM.

STADI ISTRUZIONE RISC

L'esecuzione di un'istruzione è suddivisa in vari STADI che dipendono dal tipo di architettura, ad esempio nel caso X86 si hanno circa 15 stadi. A scopo didattico noi vedremo i 5 stadi che compongono un'istruzione RISC.

1 - FETCH: L'istruzione corrente viene prelevata dalla memoria e viene calcolato l'indirizzo dell'istruzione successiva.

2 - DECODE: Eventuali operandi immediati dell'istruzione vengono letti, così come eventuali registri di input.

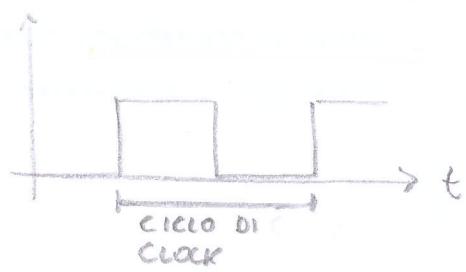
3 - EXECUTE: Se richiesto dall'istruzione, la ALU esegue un'operazione.

4 - MEMORY: Se richiesto dall'istruzione, la memoria viene acceduto in lettura o scrittura.

5 - WRITE-BACK: Se richiesto dall'istruzione, i registri di output vengono aggiornati.

La CPU per scandire il tempo e consentire l'attivazione delle varie componenti usa un clock.

Per semplicità supponiamo che la durata di uno stadio è pari ad un ciclo di clock.



Si osservi che per alcune istruzioni NON tutti gli stadi eseguono una qualche operazione poiché non richiesto.

ESEMPIO

Per vedere qualche esempio di esecuzione degli stadi di una istruzione consideriamo alcune istruzioni dell'architettura x86, si osservi che solo alcune istruzioni possono essere "inquadrate" mediante i precedenti 5 stadi poiché x86 è CISC

Si osservi che l'istruzione:

`add $1,(%eax)`

NON puo' essere eseguito poiché
è necessario l'accesso a memoria
prima dello stadio EXECUTE.

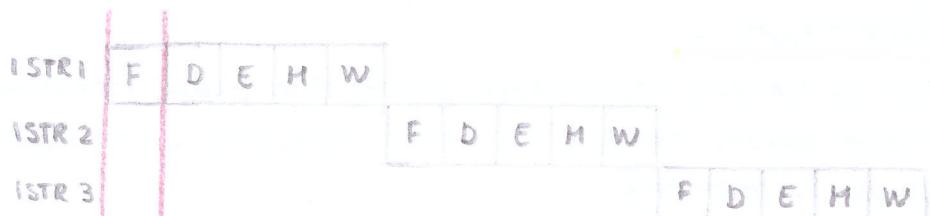
`movl $1,%eax | addl %eax,%ecx`

	<code>movl \$1,%eax</code>	<code>addl %eax,%ecx</code>
Fetch	✓	✓
Decode	✓	✓
Execute	✗	✓
Memory	✗	✗
Write Back	✓	✓

PIPELINING

L'idea del PIPELINING (Catena di montaggio) è quello di eseguire più istruzioni contemporaneamente, dato che ogni stadio è associato ad una specifica componente hardware è possibile sfruttare le componenti libere durante l'esecuzione di un'istruzione per eseguirne altre. Confrontiamo l'esecuzione di tre istruzioni con e senza pipelining:

- SENZA PIPELINING:



- CON PIPELINING



Notiamo che un sistema con pipelining consente di eseguire tre istruzioni in soli sette cicli di clock invece di 15.

Idealmente tale meccanismo consente di eseguire parallelamente un numero di istruzioni pari al numero dei diversi stadi dell'architettura

LATENZA E THROUGHPUT

Per misurare le performance di un sistema è solito introdurre due grandezze:

- LATENZA: Tempo necessario per l'esaurirsi di un evento.

Nel nostro caso rappresenta il tempo necessario ad eseguire un'istruzione.

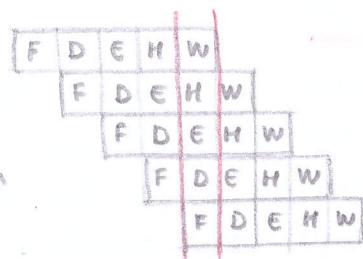
- THROUGHPUT: Numero di eventi nell'unità di tempo.

Nel nostro caso rappresenta il numero di istruzioni eseguite per unità di tempo.

Dalla precedente osservazione sul confronto tra un sistema con o senza pipelining deduciamo che quest'ultimo AUMENTA il throughput e NON influenza la latenza (potrebbe leggermente aumentare a causa della complicazione hardware).

HAZARDS

Abbiamo concluso che idealmente nella nostra architettura RISC a 5 stadi è possibile eseguire parallelamente 5 istruzioni. Tuttavia nella realtà ciò non è sempre possibile poiché vi sono diversi HAZARD (Pericoli / Ostacoli) che impediscono la realizzazione.



Hazard Strutturali

Si ha un HAZARD STRUTTURALE quando due o più istruzioni richiedono di usare simultaneamente lo stesso componente hardware come ad esempio la memoria.

Evidenziando le componenti coinvolte in ogni stadio abbiamo il seguente pipelining:

	(F)	(D)	(E)	(H)	(W)			
ISTR1	MEH	REG	ALU	HEH	REG			
ISTR2		HEH	REG	ALU	HEH	REG		
ISTR3			HEH	REG	ALU	HEH	REG	
ISTR4				HEH	REG	ALU	HEH	REG

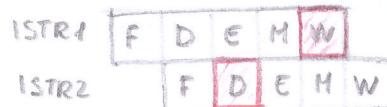
Osserviamo che al 4° ciclo di clock due istruzioni accedono allo stesso componente circolare, ovvero la memoria. Per ovviare a tale problema gli attuali processori hanno due cache L1 in modo che la memoria usata da fetch sia diversa da quella usata dallo stage memory. (Vedi Cache)

- Hazard sui Dati

Si ha un HAZARD sui DATI quando uno studio di un'istruzione dipende da un studio di un'altra istruzione e quindi non possono essere eseguite completamente in parallelo. Ciò accade ad esempio quando un'istruzione ha come input l'output di quella precedente.

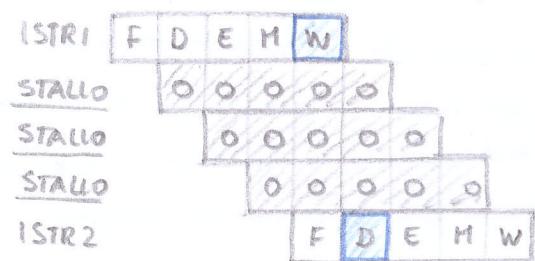
Ad esempio si considerino le seguenti istruzioni:

movl \$1, %eax
movl %eax, %ecx



In questo caso si ha un hazard sui dati poiché ISTR2 deve attendere che ISTR1 aggiorni il valore di %eax nello studio di writeback, quindi quest'ultimo deve precedere la decode di ISTR2.

Questo tipo di hazard può essere risolto dal hardware, con una conseguente penalità sul throughput, inserendo degli STALLI tra le istruzioni coinvolte.



Un altro metodo per mitigare gli hazard sui dati può essere applicato a livello software dal compilatore o programmatore ASH, e consiste nel riordinare le istruzioni in modo da inserire istruzioni "isolote" tra le due in contrasto invece di utilizzare gli stalli. Tale tecnica di ottimizzazione è detta INSTRUCTION SCHEDULING.

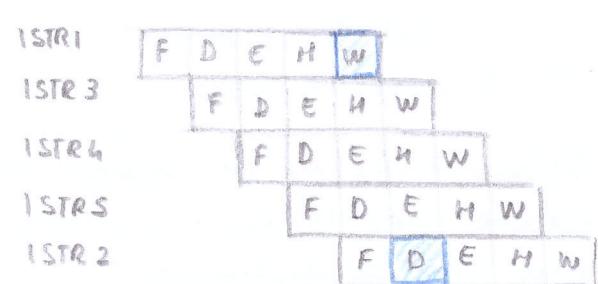
Si consideri il seguente esempio:

ISTR1	movl \$1, %eax				
ISTR2	movl %eax, %ecx	Hazard			
ISTR3	movl \$2, %esi				
ISTR4	movl \$3, %edi				
ISTR5	movl \$4, %edx				

Istruzione Independenti

ISTR1	movl \$1, %eax				
ISTR3	movl \$2, %esi				
ISTR4	movl \$3, %edi				
ISTR5	movl \$4, %edx				
ISTR2	movl %eax, %ecx				

In questo modo, risolvendo lo hazard via SW non si ha una penalità sul throughput.



- Hazard sul Controllo

Si hanno HAZARD SUL CONTROLLO quando sono presenti istruzioni di salto condizionato, poiché non conosciamo a priori l'istruzione successiva da caricare.

Consideriamo il seguente esempio:

Il valore di %eax è noto solo al completamento di ISTR1 quindi non conosciamo l'istruzione che seguirà ISTR2, potrebbe essere ISTR3 oppure ISTR4

ISTR1	testl %eax, %eax
ISTR2	je E
ISTR3	movl %ecx, %edx
ISTR4	E: addl \$1, %ebx

Un metodo inefficiente per risolvere tale hazard consiste nell'inserimento di stalli tra ISTR2 e la successiva in modo da attendere il completamento di ISTR1

Un alternativa a tale metodo consiste nel predire l'istruzione successiva avendo fatto una BRANCH PREDICTION e caricare dopo ISTR2 l'istruzione scelta.

Nel caso in cui la predizione è ERRATA (BRANCH MISPREDICTION) l'hardware provvede automaticamente a svuotare lo pipelineing tornando allo stato precedente alla predizione in modo da eseguire l'istruzione corretta.

Due strategie banali per la predizione sono:

- ALWAYS TAKEN: Prevedo sempre un salto

- NEVER TAKEN: Prevedo sempre di non saltare.

Si osservi che nel caso di salti all'indietro, associati quindi a loop, la strategia migliore è quella ALWAYS TAKEN poiché solitamente un ciclo viene eseguito più di una volta e quindi avremo alla fine una sola predizione errata.

Il compilatore per ridurre questo tipo di hazard, utilizza istruzioni BRANCHLESS che non causano la ramificazione del flusso di controllo. (CHOV, SET)

testl %eax, %eax	ISTR1	testl %eax, %eax
je E	→	ISTR2 cmove %ecx, %edx
movl %ecx, %edx		ISTR3 addl \$1, %ebx
E: addl \$1, %ebx		

Si osservi che occorre comunque attendere la terminazione di ISTR1 prima di eseguire ISTR2, tuttavia vi si possono interporre istruzioni indipendenti o stalli come visto precedentemente

MISURA DEL TEMPO IN C

Abbiamo visto come misurare il tempo di esecuzione di un programma tramite time oppure tramite gprof, vediamo ora come misurare il tempo trascorso direttamente in C ad esempio per misurare il tempo impiegato da una determinata funzione.

Nella LIBC sono presenti diverse funzioni che consentono la misurazione del tempo noi ne vediamo due.

- GETTIMEOFDAY #include <sys/time.h>

```
int gettimeofday(struct timeval *tv, struct timezone *tz)
```

La seguente funzione consente di ottenere il tempo trascorso dalla UNIX EPOCH, ovvero dal 1 Gennaio 1970, tramite il parametro tv. Il parametro tz e' deprecato, quindi lo porremo a NULL.

struct timeval

time_t tv.sec;	→ Dallo Unix epoch sono passati tv.sec secondi e tv.usec microsecondi.
suseconds_t tv.usec;	

Tale funzione non e' molto affidabile poiche' l'orologio a cui si affidano, ovvero quello di sistema, non e' monotono crescente a causa di un cambio orario da parte dell'amministratore oppure della sincronizzazione tramite il NETWORK TIME PROTOCOL (NTP) necessario per evitare il drift a cui sono soggetti gli orologi.

```

35 #include "e1.h"
36 #include <stdio.h>
37 #include <stdlib.h>
38 #include <assert.h>
39 #include <sys/time.h> // gettimeofday
40 #define NUM 5000 // provare con 5
41 // Attenzione: se l'amministratore cambia l'orologio di sistema
42 // possono esserci discontinuità nelle misurazioni
43 double get_real_time_msec() {
44     struct timeval tv;
45     gettimeofday(&tv, NULL);
46     return tv.tv_sec*1E03+tv.tv_usec*1E-03; // Tempo in milliseconds
47 }
48 int main() {
49     int i, *v = malloc(NUM*sizeof(int));
50     assert(v != NULL);
51     for (i=0; i<NUM; ++i) v[i] = NUM-i-1;
52     printf("\nstart sorting...\n-----\n");
53     double start = get_real_time_msec();
54     sort(v, NUM);
55     double elapsed = get_real_time_msec() - start;
56     for (i=0; i<10; ++i) printf("v[%d]=%d\n", i, v[i]);
57     printf("-----\n");
58     printf("Tempo richiesto da sort: %f msec\n", elapsed);
59     free(v);
60     return EXIT_SUCCESS;

```

• CLOCK_GETTIME

Una funzione migliore rispetto a `gettimeofday()` è `CLOCK_GETTIME` la quale ha una precisione maggiore (ns) ed inoltre può affidarsi a diversi orologi.

```
#include <time.h>
int clock_gettime(clockid_t clk_id, struct timespec *tp)
```

Il primo parametro `clk_id` serve per specificare l'orologio da utilizzare:

- CLOCK_REALTIME: Orologio di sistema, rappresenta il tempo trascorso dalla Unix epoch. Soffre degli stessi problemi di `gettimeofday()`.
- CLOCK_REALTIME_COARSE: Stesso orologio di `CLOCK_REALTIME` ma la funzione `clock_gettime()` ha una velocità maggiore a penalizzata tuttavia la precisione. (Disponibile da UNUX 2.6.32)
- CLOCK_MONOTONIC: Orologio NON reimpostabile e garantito monotono crescente.
- CLOCK_MONOTONIC_COARSE: Equivale a `CLOCK_REALTIME_COARSE`

Il secondo parametro invece rappresenta la struct in cui verrà memorizzato il tempo in modo simile a `gettimeofday()`, ma con precisione di ns (invece di `µs`)

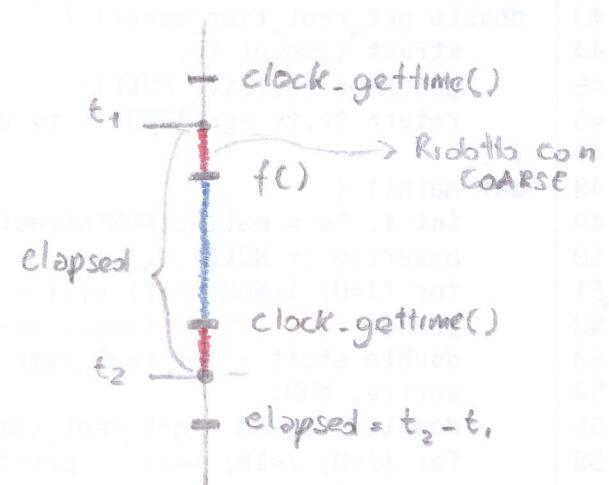
```
struct timespec {
    time_t tv_sec // second
    long tv_nsec // nanoseconds
}; // msec = tv.sec * 103 + tv_nsec * 10-6
```

- Clock Coarse

Gli orologi di tipo coarse sono utili quando si vuole misurare una piccola differenza temporale ed evitare che il tempo di esecuzione di `clock_gettime()` influenzi eccessivamente la misura.

Consideriamo ad esempio di voler misurare il tempo di esecuzione di una funzione `f()`.

Notiamo che nella misura di elapsed si considera necessariamente una parte di `clock_gettime()`, gli orologi di tipo COARSE riducono questa parte andando però a penalizzare la precisione.



Esempio (clock_gettime())

```
7 #include "e2.h"
8 #include <stdio.h>
9 #include <stdlib.h>
10 #include <assert.h>
11 #include <time.h> // clock_gettime
12 #define NUM 5 // provare con 5
13 double get_real_time_msec() {
14     struct timespec ts;
15     clock_gettime(CLOCK_MONOTONIC, &ts);
16     return ts.tv_sec*1E03 + ts.tv_nsec*1E-06;
17 }
18 int main() {
19     int i, *v = malloc(NUM*sizeof(int));
20     assert(v != NULL);
21     for (i=0; i<NUM; ++i) v[i] = NUM-i-1;
22     printf("\nstart sorting...-----\n");
23
24     double start = get_real_time_msec();
25     sort(v, NUM);
26     double elapsed = get_real_time_msec() - start;
27
28     for (i=0; i<10; ++i)
29         printf("v[%d]=%d\n", i, v[i]);
30     printf("-----\n");
31     printf("Tempo richiesto da sort: %f msec\n", elapsed);
32     free(v);
33     return EXIT_SUCCESS;
34 }
```

CONFRONTO OROLOGI

Precedentemente abbiamo visto a cosa servono i **clock COARSE**, confrontiamo ora il tempo in ns, tramite `clock_gettime()`, necessario all'esecuzione dei tre tipi di orologi visti.

Si osservi che nel codice eseguiamo le funzioni N volte e poi dividiamo l'elapsed totale in modo da ottenere una media delle singole esecuzioni.

L'output ottenuto con un Intel i5 (2011) è il seguente, si noti la differenza sostanziale del `clock coarse`.

```
51 -----
52 Measuring gettimeofday...
53 Time per gettimeofday: 31.185272 nsec
54 -----
55 Measuring clock_gettime...
56 Time per clock_gettime: 26.416832 nsec
57 -----
58 Measuring clock_gettime [coarse]...
59 Time per clock_gettime: 8.382228 nsec
60 -----
```

Il codice per il confronto e' il seguente:

```
7 #include <sys/time.h> // gettimeofday
8 #include <time.h> // clock_gettime
9 #define N 1000000
10 long get_real_time_nsec() {
11     struct timespec ts;
12     clock_gettime(CLOCK_MONOTONIC, &ts);
13     return ts.tv_sec*1000000000+ts.tv_nsec;}
14 void measure_usec(unsigned n) {
15     int i; struct timeval tv;
16     printf("-----\n");
17     printf("Measuring gettimeofday...\n");
18     long start = get_real_time_nsec();
19     for (i=0; i+3 < n; i+=4) {
20         gettimeofday(&tv, NULL);
21         gettimeofday(&tv, NULL);
22         gettimeofday(&tv, NULL);
23         gettimeofday(&tv, NULL);}
24     long elapsed = get_real_time_nsec()-start;
25     printf("Time per gettimeofday: %f nsec\n", (double)elapsed/n);}
26 void measure_nsec(unsigned n) {
27     int i; struct timespec ts;
28     printf("-----\n");
29     printf("Measuring clock_gettime...\n");
30     long start = get_real_time_nsec();
31     for (i=0; i+3 < n; i+=4) {
32         clock_gettime(CLOCK_MONOTONIC, &ts);
33         clock_gettime(CLOCK_MONOTONIC, &ts);
34         clock_gettime(CLOCK_MONOTONIC, &ts);
35         clock_gettime(CLOCK_MONOTONIC, &ts);}
36     long elapsed = get_real_time_nsec()-start;
37     printf("Time per clock_gettime: %f nsec\n", (double)elapsed/n);}
38 #ifdef __linux__ // Since Linux 2.6.32; Linux-specific
39 void measure_nsec_coarse(unsigned n) {
40     int i; struct timespec ts;
41     printf("-----\n");
42     printf("Measuring clock_gettime [coarse]...\n");
43     long start = get_real_time_nsec();
44     for (i=0; i+3 < n; i+=4) {
45         clock_gettime(CLOCK_MONOTONIC_COARSE, &ts);
46         clock_gettime(CLOCK_MONOTONIC_COARSE, &ts);
47         clock_gettime(CLOCK_MONOTONIC_COARSE, &ts);
48         clock_gettime(CLOCK_MONOTONIC_COARSE, &ts);    }
49     long elapsed = get_real_time_nsec()-start;
50     printf("Time per clock_gettime: %f nsec\n", (double)elapsed/n);}
51 #endif
52 int main() {
53     measure_usec(N);
54     measure_nsec(N);
55     #ifdef __linux__
56     measure_nsec_coarse(N);
57     #endif
58     return EXIT_SUCCESS;
59 }
60
```