

Sistemi di Calcolo

Ultimo aggiornamento: 12 giugno 2019



Camil Demetrescu, Emilio Coppa, Daniele Cono D'Elia

*Dipartimento di Ingegneria Informatica, Automatica e Gestionale "A. Ruberti"
Sapienza Università di Roma*

Indice

[1 Cosa è un sistema di calcolo?](#)

[1.1 CPU](#)

[1.2 Bus](#)

[1.3 I/O bridge](#)

[1.4 Controller e adattatori](#)

[2 Come viene programmato un sistema di calcolo?](#)

[2.1 Linguaggi di programmazione di alto livello e di basso livello](#)

[2.2 Compilazione vs. interpretazione](#)

[2.1 Stadi della compilazione di un programma C](#)

[2.1.1 Il toolchain di compilazione gcc](#)

[2.1.2 Programmi in formato testo, oggetto, ed eseguibile \(ELF, Mach-O\)](#)

[2.1.2 Disassemblare programmi in codice macchina: objdump](#)

[3 Come viene tradotto in linguaggio macchina un programma C?](#)

[3.1 Instruction set architecture \(ISA\) IA32 - Sintassi AT&T](#)

[3.1.1 Tipi di dato macchina](#)

[3.1.2 Corrispondenza tra tipi di dato C e tipi di dato macchina](#)

[3.1.3 Registri](#)

[3.1.4 Operandi e modi di indirizzamento della memoria](#)

[3.1.5 Rappresentazione dei numeri in memoria: big-endian vs. little-endian](#)

[3.1.6 Istruzioni di movimento dati](#)

[3.1.6.1 Stessa dimensione sorgente e destinazione: MOV](#)

[3.1.6.2 Dimensione destinazione maggiore di quella sorgente: MOVZ, MOVS](#)

[3.1.6.3 Movimento dati da/verso la stack: PUSH, POP](#)

[3.1.7 Istruzioni aritmetico-logiche](#)

[3.1.7.1 L'istruzione LEA \(load effective address\)](#)

[3.1.7.2 Istruzioni di shift SHL, SHR, SAL, SAR](#)

[3.1.8 Istruzioni di salto](#)

[3.1.8.1 Salti incondizionati: JMP](#)

[3.1.8.2 Salti condizionati e condition code: Jcc, CMP](#)

[3.1.8.3 Chiamata e ritorno da funzione: CALL e RET](#)

[3.1.9 Altre istruzioni e vincoli sulle istruzioni](#)

[3.1.9.1 Istruzioni di assegnamento condizionato: CMOVcc](#)

[3.1.9.2 Altre istruzioni di confronto: TEST](#)

[3.1.9.3 Altre istruzioni di assegnamento: SETcc](#)

[3.1.9.4 Vincoli sulle istruzioni](#)

[3.2 Traduzione dei costrutti C in assembly IA32](#)

[3.2.1 Istruzioni condizionali](#)

[3.2.1.1 Istruzione if](#)
[3.2.1.1 Istruzione if...else](#)

[3.2.2 Cicli](#)

[3.2.2.1 Istruzione while](#)
[3.2.2.1 Istruzione for](#)

[3.2.3 Funzioni](#)

[3.2.3.1 Restituzione valore](#)
[3.2.3.2 Stack frame e registro EBP](#)
[3.2.3.3 Passaggio dei parametri](#)
[3.2.3.4 Registri caller-save e callee-save](#)
[3.2.3.5 Variabili locali](#)

[3.2.4 Array e aritmetica dei puntatori](#)

[4 Come vengono eseguiti i programmi?](#)

[4.1 Processi](#)

[4.2 Esecuzione dei programmi](#)

[4.2.1 Esecuzione di una singola istruzione](#)
[4.2.2 Esecuzione simultanea di più istruzioni: pipelining](#)
[4.2.3 Esecuzione simultanea di più processi](#)
[4.2.3.1 Stati di un processo, schedulazione, preemption, time-sharing](#)
[4.2.3.2 Creazione di processi: fork](#)
[4.2.3.3 Recupero del pid dei processi: getpid, getppid](#)
[4.2.3.4 Attesa della terminazione di un figlio: wait](#)
[4.2.3.5 Caricamento di programmi: exec e invocazione del loader](#)

[4.3 Flusso del controllo eccezionale](#)

[4.3.1 Interruzioni](#)

[4.3.1.1 Preemption e context switch tra processi](#)
[4.3.1.2 System call](#)
[4.3.1.3 Passaggio di stato di un processo da waiting a ready: interrupt vs. polling](#)

[4.3.2 Segnali](#)

[4.3.2.1 Definizione segnali](#)
[4.3.1.2 Invio segnali](#)
[4.3.1.4 Gestione segnali: sigaction](#)
[4.3.1.3 Attesa segnali: pause](#)
[4.3.1.4 Timer](#)

[5 Come vengono gestiti i dati in memoria centrale?](#)

[5.1 Come viene allocata la memoria?](#)

[5.1.2 Allocazione dinamica della memoria](#)

[5.1.2.1 Frammentazione interna ed esterna](#)
[5.1.2.2 Qualità di un allocatore: tempo e spazio](#)
[5.1.2.3 Allocazione in cascata della memoria](#)

- [5.1.1 Memoria fisica e memoria virtuale](#)
- [5.1.3 Allocazione nella memoria fisica: memoria virtuale](#)
 - [5.1.3.1 Mapping tra indirizzi virtuali e indirizzi fisici: MMU](#)
 - [5.1.3.2 Paginazione](#)
 - [5.1.3.3 Come avviene il mapping degli indirizzi logici su quelli fisici?](#)
 - [5.1.3.4 Paginazione con bit di validità](#)
- [5.1.4 Allocazione nella memoria logica: malloc e free](#)
- [5.2 Come ottenere le migliori prestazioni usando la memoria?](#)
 - [5.2.1 Memorie cache](#)
 - [5.2.2 Politiche di rimpiazzo delle linee](#)
 - [5.2.3 Località spaziale e temporale](#)
 - [5.2.4 Esempi](#)
 - [5.2.4.1 Somma di matrice](#)
 - [5.2.4.2 Analisi di una sequenza di accessi](#)
 - [5.2.5 Associatività](#)
 - [5.2.6 Tipi di cache miss](#)
 - [5.2.7 Gerarchie di memoria](#)
- [6 Come vengono ottimizzati i programmi?](#)
 - [6.1 Quanto è importante ottimizzare le prestazioni?](#)
 - [6.2 Tecniche di ottimizzazione delle prestazioni di un programma](#)
 - [6.2.1 Livelli di ottimizzazione in gcc](#)
 - [6.2.2 Ridurre il numero di istruzioni eseguite](#)
 - [6.2.2.1 Constant folding](#)
 - [6.2.2.2 Constant propagation](#)
 - [6.2.2.3 Common subexpression elimination](#)
 - [6.2.2.4 Dead code elimination](#)
 - [6.2.2.5 Loop-invariant code motion \(hoisting\)](#)
 - [6.2.2.6 Function inlining](#)
 - [6.2.3 Ridurre il costo delle istruzioni eseguite](#)
 - [6.2.3.1 Register allocation](#)
 - [6.2.3.2 Strength reduction](#)
 - [6.2.3.3 Riduzione dei cache miss](#)
 - [6.2.3.4 Allineamento dei dati in memoria](#)
 - [6.2.4 Ridurre lo spazio di memoria](#)
 - [6.2.4.1 Ottimizzare lo spazio richiesto dalle strutture C](#)
 - [6.2.5 Le ottimizzazioni dei compilatori sono le migliori possibili?](#)
 - [6.3 Quali parti di un programma ottimizzare?](#)
 - [6.3.1 Scala degli eventi in un sistema di calcolo](#)
 - [6.3.2 Speedup e legge di Amdahl](#)
 - [6.3.2 Profilazione delle prestazioni](#)
 - [6.3.2.1 System call di misurazione del tempo](#)

6.3.2.2 gprof

Bibliografia

Appendice A: tabella dei caratteri ASCII a 7 bit

A.1 Caratteri ASCII di controllo

A.2 Caratteri ASCII stampabili

Appendice B: il file system

B.1.1 L'albero delle directory

B.1.2 Percorso assoluto e percorso relativo

Appendice C: la shell dei comandi

C.0 Invocazione di un comando

C.1 Manipolazione ed esplorazione del file system

C.1.1 pwd: visualizza il percorso assoluto della directory corrente

C.1.2 cd: cambia directory corrente

C.1.3 ls: elenca il contenuto di una directory

C.1.4 touch: crea un file vuoto o ne aggiorna la data di modifica

C.1.5 mv: rinomina o sposta un file o una directory

C.1.6 mkdir: crea una nuova directory vuota

C.1.7 rmdir: elimina una directory, purché sia vuota

C.1.8 rm: elimina un file o una directory

C.1.9 cp: copia un file o una directory

C.2 Altri comandi utili

C.3 Permessi UNIX e il comando chmod

C.4 Redirezione dello stdin, stdout, e stderr di un processo

C.4 Esecuzione in background e terminazione di un processo

Appendice D: Debugging di una applicazione C

D.1 Compilazione di un programma C con informazioni di debugging

D.2 Valgrind: identificazione di memory leak ed accessi alla memoria non validi

1 Cosa è un sistema di calcolo?

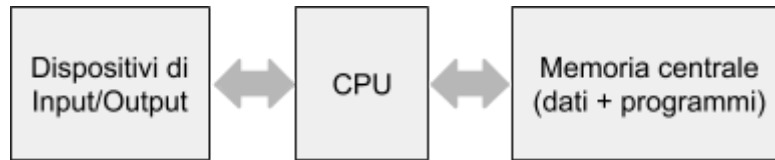
Un **sistema di calcolo** consiste di **software** e **hardware** che operano insieme per supportare l'esecuzione di programmi. Esempi di sistemi di calcolo sono gli smartphone, i tablet, i computer fissi e i portatili, i data center che gestiscono i nostri account Facebook, Twitter o Google, i supercomputer usati dal CERN di Ginevra per simulare la fisica delle particelle, ma anche i televisori di nuova generazione che consentono di navigare in Internet, i riproduttori multimediali, i modem/router che usiamo a casa per connetterci alla rete ADSL, le macchine fotografiche digitali, i computer di bordo delle automobili, le console per i videogiochi (PlayStation, Wii, Xbox, ecc.), e molto altro ancora che non sospetteremmo possa essere pensato come un sistema di calcolo.



In generale, un sistema di calcolo è qualsiasi **sistema programmabile**, cioè in grado di eseguire compiti diversi in base alle istruzioni fornite da un **programma**. Un sistema di calcolo può essere formato da un **singolo nodo**, cioè un insieme di parti hardware strettamente connesse tra loro e spazialmente adiacenti, oppure da **più nodi connessi mediante una rete di comunicazione**.

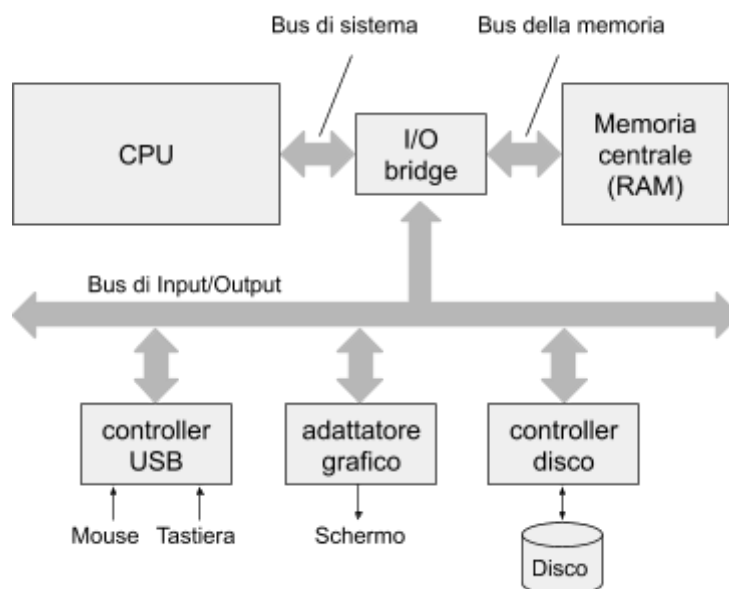
Sono sistemi di calcolo a singolo nodo i computer fissi e portatili, gli smartphone, i tablet, ecc. Esempi di sistemi multi-nodo sono i data center usati dai grandi provider come Facebook, Twitter e Google per gestire i loro sistemi di social networking e i supercomputer, in cui più nodi di calcolo sono connessi da una rete di comunicazione ad alta velocità (es. Infiniband). Questo tipo di sistema viene anche detto **cluster**.

In questo corso tratteremo sistemi di calcolo in cui l'hardware dei singoli nodi è organizzato secondo il **modello di Von Neumann**:



La **memoria centrale** contiene **dati da elaborare e programmi**, i **dispositivi di input e output** scambiano dati e interagiscono con il mondo esterno, mentre la **CPU** (Central Processing Unit, o Unità Centrale di Elaborazione) esegue le **istruzioni** di un programma.

Più in dettaglio, l'organizzazione tipica di un calcolatore moderno è la seguente:



Il diagramma fornisce in maggiore dettaglio le componenti architetture tipiche di un sistema di calcolo a singolo nodo, descritte nei paragrafi seguenti.

1.1 CPU

La **CPU** (o **microprocessore**) è il cuore del sistema che esegue programmi scritti in **linguaggio macchina** (codice **nativo**), rappresentati come sequenze di byte che codificano istruzioni. Il **set di istruzioni**, cioè l'insieme delle istruzioni riconosciute dalla CPU è specifico alla particolare famiglia di CPU, e può differire anche sostanzialmente fra modelli diversi prodotti da aziende diverse¹. Le istruzioni vengono eseguite dalla CPU nell'ordine in cui appaiono in memoria ed effettuano tipicamente operazioni che:

¹ Esempi di famiglie di microprocessori sono [x86](#) (Intel, AMD), [PowerPC](#) (Apple-Mototola-IBM), [SPARC](#) (Sun Microsystems), [ARM](#) (ARM Holdings).

- A. **trasferiscono dati** all'interno della CPU, all'interno della memoria, fra memoria e CPU, e fra dispositivi esterni e la CPU.
- B. calcolano operatori **aritmetico/logici** (somme, prodotti, ecc.), operatori **booleani** (congiunzione, disgiunzione, negazione, ecc.), operatori **relazionali** (uguale, maggiore, minore, ecc.).
- C. effettuano **salti** che permettono di continuare l'esecuzione non dall'istruzione successiva in memoria, ma da un altro punto del programma. Queste istruzioni servono per realizzare cicli (for, while, ecc.) e costrutti di selezione (if ... else).

1.2 Bus

Sono strutture di interconnessione che collegano le varie componenti del sistema consentendo lo scambio dei dati. I bus sono canali di comunicazione che trasferiscono i dati tipicamente a blocchi fissi di byte, chiamati **word**. La dimensione di una word trasferita su un bus è generalmente di 4 byte (32 bit) oppure 8 byte (64 bit), a seconda dell'architettura. In questa dispensa considereremo word di 8 byte (architettura a 64 bit) e assumeremo che su un bus viene trasferita una word alla volta.

1.3 I/O bridge

Si occupa di coordinare lo scambio dei dati fra la CPU e il resto del sistema.

1.4 Controller e adattatori

Interfacciano il sistema verso il mondo esterno, ad esempio acquisendo i movimenti del mouse o i tasti premuti sulla tastiera. La differenza fra i controller e adattatore è che i controller sono saldati sulla **scheda madre** (cioè sul circuito stampato che ospita CPU e memoria e bus) oppure sono integrati nel dispositivo esterno, mentre gli adattatori sono schede esterne collegate alla scheda madre (es. adattatore video o di rete).

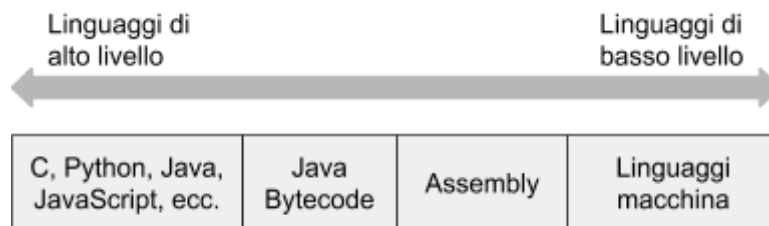
2 Come viene programmato un sistema di calcolo?

2.1 Linguaggi di programmazione di alto livello e di basso livello

Il linguaggio macchina è pensato per essere eseguito dalla CPU, ma è assolutamente inadatto per programmare. I programmatori scrivono invece i loro programmi in **linguaggi di alto livello**² come C, Python, Java, ecc., che forniscono costrutti molto più potenti, sono più semplici da imparare e usare, sono più facilmente manutenibili, ed è più semplice identificare e correggere gli errori (**debugging**).

² Un linguaggio viene chiamato di **alto livello** quando i suoi costrutti sono lontani da quelli del linguaggio macchina, e di **basso livello** altrimenti. Vi sono varie gradazioni intermedie: ad esempio, i costrutti del C sono più vicini a quelli della macchina rispetto ad alcuni costrutti Java o Python.

I programmatori professionisti interessati a scrivere codice particolarmente ottimizzato per le prestazioni oppure parti di sistemi operativi usano il linguaggio **assembly** (o assembler), un linguaggio di basso livello che descrive le istruzioni macchina utilizzando una sintassi comprensibile allo sviluppatore. I linguaggi assembly sono "traslitterazioni" dei corrispondenti linguaggi macchina che associano a ogni codice di istruzione binario un corrispondente codice mnemonico, più leggibile per un programmatore.



2.2 Compilazione vs. interpretazione

Per poter eseguire un programma scritto in un linguaggio di alto livello ci sono vari approcci possibili:

Approccio	Compilazione	Interpretazione	Ibrido: compilazione e interpretazione
Descrizione	Il programma di alto livello viene tradotto in codice macchina (nativo) in modo da poter essere eseguito direttamente dalla CPU. In generale, il processo di traduzione da un linguaggio all'altro viene chiamato compilazione .	Il programma di alto livello viene direttamente eseguito da un programma chiamato interprete , senza bisogno di essere prima compilato. I linguaggi di alto livello interpretati vengono generalmente chiamati linguaggi di scripting .	Il programma di alto livello viene prima compilato in un codice scritto in un linguaggio di livello intermedio . Il programma di livello intermedio viene poi interpretato.
Esempi linguaggi	C, C++, Fortran	Python, Perl, PHP, JavaScript, Java bytecode	Java
Vantaggi	Il compilatore genera codice ottimizzato per la piattaforma di calcolo considerata, ottenendo le migliori prestazioni possibili per un programma di alto livello.	Il programma è portabile , essendo eseguibile su qualsiasi piattaforma su cui sia disponibile un interprete per il linguaggio in cui è scritto. Inoltre, è possibile eseguirlo direttamente senza bisogno di compilarlo, rendendo più agile lo sviluppo.	Il programma è portabile , essendo eseguibile su qualsiasi piattaforma su cui sia disponibile un interprete per il linguaggio intermedio in cui è stato compilato.
Svantaggi	Il programma compilato non è portabile , essendo eseguibile solo sulla piattaforma di calcolo per cui è stato compilato	L'esecuzione interpretata è tipicamente più lenta di quella nativa ottenuta mediante un compilatore che genera codice macchina.	Vedi interpretazione.

		<p>Tuttavia, alcuni interpreti, come quello del linguaggio Java Bytecode, utilizzano una tecnica chiamata Just-in-time compilation (JIT), che consiste nel compilare il programma da eseguire in codice nativo subito prima di essere eseguito.</p>	
--	--	--	--

2.1 Stadi della compilazione di un programma C

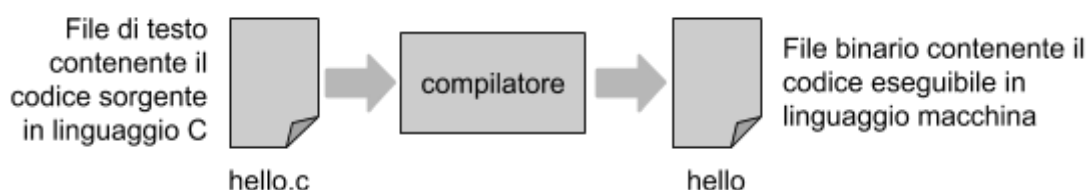
In questo corso ci concentreremo sul linguaggio C. La compilazione di un programma C può essere scomposta nella compilazione di più moduli C (anche dette “translation unit”), ciascuno residente in un file di testo con estensione “.c”. Nei casi più semplici, un programma C è formato da un’unica translation unit:

hello.c

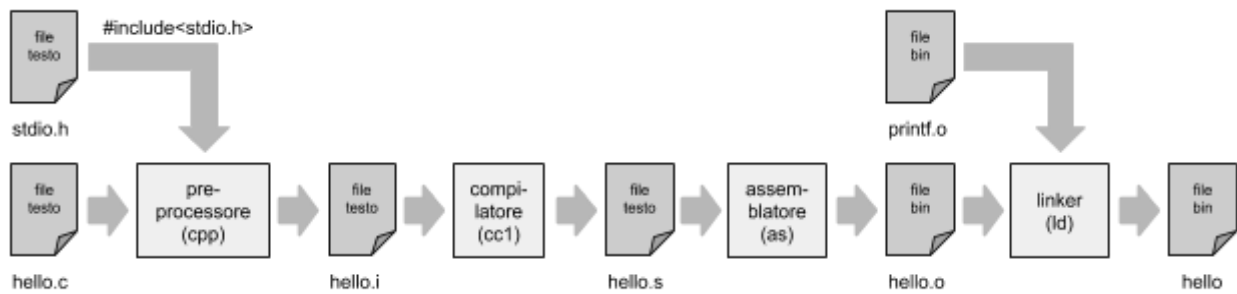
```
#include<stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

La compilazione parte dal file `hello.c` e genera un file eseguibile `hello`:



Il processo di compilazione di un programma C è in realtà composto da diversi stadi, che vengono normalmente effettuati dal sistema di compilazione (*compilation toolchain*, in inglese) senza che ce ne accorgiamo:



Gli stadi coinvolgono diversi sottoprogrammi che vengono attivati dal sistema di compilazione:

1. **Preprocessore**: prende un testo C e lo converte in un altro testo C dove le direttive `#include`, `#define`, ecc. sono state elaborate. Il testo risultante sarà un programma C senza le direttive `#`. Il preprocessore includerà nel file generato il contenuto di tutti i file che vengono specificati dalle direttive `#include`. Nel nostro esempio, oltre a leggere `hello.c`, il preprocessore leggerà anche il file `stdio.h`, disponibile nelle directory di sistema del compilatore.
2. **Compilatore**: prende un testo C senza direttive `#`, ne verifica la correttezza sintattica, e lo traduce in codice in linguaggio assembly per la piattaforma per cui si sta compilando.
3. **Assemblatore**: prende un programma scritto in assembly e lo traduce in codice macchina, generando un **file oggetto** (binario).
4. **Linker**: prende vari file oggetto e li collega insieme a formare un unico file eseguibile. Nel nostro esempio, verranno collegati ("linkati") insieme `hello.o`, che contiene il programma in codice macchina, e `printf.o`, che contiene il codice macchina che realizza la funzione `printf` invocata dal programma. Il risultato della compilazione è il file eseguibile `hello`.

2.1.1 Il toolchain di compilazione `gcc`

Linux/MacOS X

Il toolchain di compilazione `gcc` contiene vari programmi che realizzano i diversi stadi della compilazione:

1. `cpp`: preprocessore
2. `cc1`: compilatore C
3. `as`: assemblatore
4. `ld`: linker

Normalmente non si invocano questi programmi direttamente, ma i vari stadi possono essere effettuati separatamente mediante l'inclusione di opportune opzioni ("switch" della forma `-option`, dove `option` può essere: E, S, c, o) nella riga di comando con cui si invoca `gcc`:

1. `gcc -E hello.c > hello.i`: preprocessa `hello.c` e genera il programma C preprocessato `hello.i`, in cui le direttive `#` sono state elaborate. Si noti che `gcc -E hello.c` stamperebbe il testo preprocessato a video. L'uso della redirectione `> hello.i` dirotta invece l'output del preprocessore nel file `hello.i`.
2. `gcc -S hello.i`: compila il programma preprocessato `hello.i` traducendolo in un

file assembly `hello.s`.

3. `gcc -c hello.s`: assembla il file `hello.s`, scritto in linguaggio assembly, generando un file oggetto `hello.o`.
4. `gcc hello.o -o hello`: collega il file oggetto `hello.o` con i moduli della libreria di sistema (ad esempio quella contenente il codice della funzione `printf`) e genera il file eseguibile `hello`.

I passi elencati ai punti 1-4 sopra generano il file eseguibile `hello` a partire da un file sorgente `hello.c` creando uno ad uno tutti i file relativi ai vari stadi intermedi della compilazione (`hello.i`, `hello.s`, `hello.o`). Si noti che il comando:

```
gcc hello.c -o hello
```

è del tutto equivalente, con l'unica differenza che i file intermedi vengono creati come file temporanei nascosti al programmatore e poi eliminati automaticamente da `gcc`.

2.1.2 Programmi in formato testo, oggetto, ed eseguibile (ELF, Mach-O)

Linux/macOS X

Si può esaminare la natura dei vari file coinvolti nel processo di compilazione usando il comando `file`. Questo è l'output che si otterrebbe su sistema operativo Linux a 64 bit:

```
$ file hello.c
hello.c: C source, ASCII text

$ file hello.i
hello.i: C source, ASCII text

$ file hello.s
hello.s: ASCII text

$ file hello.o
hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not
stripped

$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=0x6c7467509edcd8c76549721c01468b480a6988f4, not
stripped
```

Si noti che i file `hello.c`, `hello.i` e `hello.s` sono file di testo (ASCII text), mentre i file `hello.o` e `hello` sono file binari. In particolare, gli ultimi due usano il formato binario [ELF](#) che rappresenta tutte le informazioni di un programma in codice macchina. Il formato ELF è usato da numerosi altri sistemi oltre a Linux, fra cui le console PlayStation e alcuni smartphone.

Il sistema operativo macOS X (ma anche iOS, usato su iPhone e iPad) usa invece il formato

binario [Mach-O](#) per rappresentare file oggetto ed eseguibili. Il seguente risultato è ottenuto su sistema operativo MacOS X a 64 bit:

```
$ file hello.c
hello.c: ASCII c program text

$ file hello.i
hello.i: ASCII c program text

$ file hello.s
hello.s: ASCII assembler program text

$ file hello.o
hello.o: Mach-O 64-bit object x86_64

$ file hello
hello: Mach-O 64-bit executable x86_64
```

Vi sono numerosi altri formati per i file oggetto ed eseguibili dipendenti dalla particolare piattaforma utilizzata³.

Si noti che un file in formato ELF non può essere eseguito o linkato su MacOS X. Allo stesso modo, un file in formato Mach-O non può essere eseguito o linkato in Linux⁴. Tutto questo indipendentemente dall'hardware soggiacente, che potrebbe anche essere lo stesso. Pertanto, **i formati eseguibili non garantiscono in genere la portabilità dei programmi.**

2.1.2 Disassemblare programmi in codice macchina: `objdump`

Disassemblare un programma in **linguaggio macchina** consiste nel tradurne le istruzioni nel codice **assembly** corrispondente in modo che possano essere analizzate da un programmatore. E' l'operazione inversa a quella dell'assemblamento.

Linux: `objdump -d` (**disassemblato**)

In ambiente Linux è possibile disassemblare un file oggetto o un file eseguibile usando il comando `objdump -d`.

Consideriamo il seguente semplice modulo `somma.c`:

```
int somma(int x, int y) {
    return x + y;
}
```

Compilandolo otteniamo un file oggetto `somma.o`:

³ http://en.wikipedia.org/wiki/Comparison_of_executable_file_formats

⁴ <http://unix.stackexchange.com/questions/3322/what-makes-osx-programs-not-runnable-on-linux>

```
$ gcc -c somma.c
```

che possiamo disassemblare come segue:

```
$ objdump -d somma.o
```

Il comando `objdump` invia l'output sul canale standard (di default è il terminale):

```
somma.o:  file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <somma>:
 0:  55                      push   %rbp
 1:  48 89 e5                mov    %rsp,%rbp
 4:  89 7d fc                mov    %edi,-0x4(%rbp)
 7:  89 75 f8                mov    %esi,-0x8(%rbp)
 a:  8b 45 f8                mov    -0x8(%rbp),%eax
 d:  8b 55 fc                mov    -0x4(%rbp),%edx
10:  01 d0                  add    %edx,%eax
12:  5d                      pop    %rbp
13:  c3                      retq

          Codice
          macchina

          Codice
          assembly

          Spiazzamento (offset) all'interno della sezione
```

Si noti che l'output riporta per ogni funzione (in questo caso la sola `somma`):

1. lo spiazzamento (offset) dell'istruzione all'interno della sezione (in esadecimale)
2. il codice macchina (in esadecimale)
3. il corrispondente codice assembly (usando i nomi mnemonici delle istruzioni).

Nel nostro esempio, il codice macchina è della famiglia x86 e come si vede le istruzioni occupano un numero variabile di byte (ad esempio, il codice macchina dell'istruzione assembly `retq` è il byte `c3` e quello dell'istruzione assembly `mov %rsp,%rbp` sono i tre byte `48, 89 ed e5`).

Un **disassemblato misto** contiene il codice sorgente inframmezzato a quello macchina/assembly, facilitando al programmatore l'analisi del codice.

Linux: `objdump -S` (**disassemblato misto**)

Se il programma è stato compilato con l'opzione `-g` di `gcc`, usando il comando `objdump -S` è possibile ottenere un **disassemblato misto** in cui il codice sorgente e quello macchina/assembly sono inframmezzati:

```
$ gcc -g -c somma.c
$ objdump -S somma.o

somma.o:  file format elf64-x86-64
```

Disassembly of section .text:

0000000000000000 <somma>:

int somma(int x, int y){

0:	55	push	%rbp
1:	48 89 e5	mov	%rsp,%rbp
4:	89 7d fc	mov	%edi,-0x4(%rbp)
7:	89 75 f8	mov	%esi,-0x8(%rbp)
	return x+y;		
a:	8b 45 f8	mov	-0x8(%rbp),%eax
d:	8b 55 fc	mov	-0x4(%rbp),%edx
10:	01 d0	add	%edx,%eax
}			
12:	5d	pop	%rbp
13:	c3	retq	

Ad esempio, le istruzioni macchina/assembly con offset compreso tra a e 12 (escluso) sono la traduzione dell'istruzione `return x+y` del programma C.

3 Come viene tradotto in linguaggio macchina un programma C?

I sistemi di calcolo si basano su un certo numero di **astrazioni** che forniscono una visione più semplice del funzionamento della macchina, nascondendo dettagli dell'implementazione che possono essere, almeno in prima battuta, ignorati.

Due delle più importanti astrazioni sono:

- La **memoria**, vista come un grosso array di byte.
- L'**instruction set architecture** (ISA), che definisce:
 - a. lo stato della CPU;
 - b. il formato delle sue istruzioni;
 - c. l'effetto che le istruzioni hanno sullo stato.

Per tradurre codice di alto livello (ad esempio in linguaggio C) in codice macchina, i compilatori si basano sulla descrizione astratta della macchina data dalla sua ISA.

Due delle ISA più diffuse sono:

- IA32, che descrive le architetture della famiglia di processori x86 a 32 bit;
- x86-64, che descrive le architetture della famiglia di processori x86 a 64 bit.

L'x86-64 è ottenuto come estensione dell'IA32, con cui è **retrocompatibile**. Le istruzioni IA32 sono infatti presenti anche nell'x86-64, ma l'x86-64 introduce **nuove istruzioni** non supportate dall'IA32. Programmi scritti in linguaggio macchina per piattaforme IA32 possono essere eseguiti anche su piattaforme x86-64, ma in generale non vale il viceversa. In questa dispensa tratteremo l'IA32.

3.1 Instruction set architecture (ISA) IA32 - Sintassi AT&T

3.1.1 Tipi di dato macchina

L'IA32 ha sei tipi di dato numerici primitivi (tipi di dato macchina):

Tipo di dato macchina	Rappresen- tazione	Suffisso assembly	Dimensione in byte
Byte	intero	b	1
Word	intero	w	2
Double word	intero	l	4
Single precision	virgola mobile	s	4
Double precision	virgola mobile	l	8

Extended precision	virgola mobile	t	12 (10 usati)
--------------------	----------------	---	---------------

I tipi macchina permettono di rappresentare sia **numeri interi** che **numeri in virgola mobile**. Si noti che il tipo Extended precision richiede 12 byte in IA32. Tuttavia, di questi solo 10 byte (80 bit) sono effettivamente usati.

Ogni tipo ha un corrispondente **suffisso assembly** che, come vedremo, viene usato per denotare il **tipo degli operandi di una istruzione**.

3.1.2 Corrispondenza tra tipi di dato C e tipi di dato macchina

La seguente tabella mostra la corrispondenza tra i tipi di dato primitivi C (interi, numeri in virgola mobile e puntatori) e i tipi di dato primitivi macchina:

Tipo di dato C	Tipo di dato macchina	Suffisso	Dimensione in byte
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Double word	l	4
long long	<i>non supportato</i>	-	8
puntatore	Double word	l	4
float	Single precision	s	4
double	Double precision	l	8
long double	Extended precision	t	12 (10 usati)

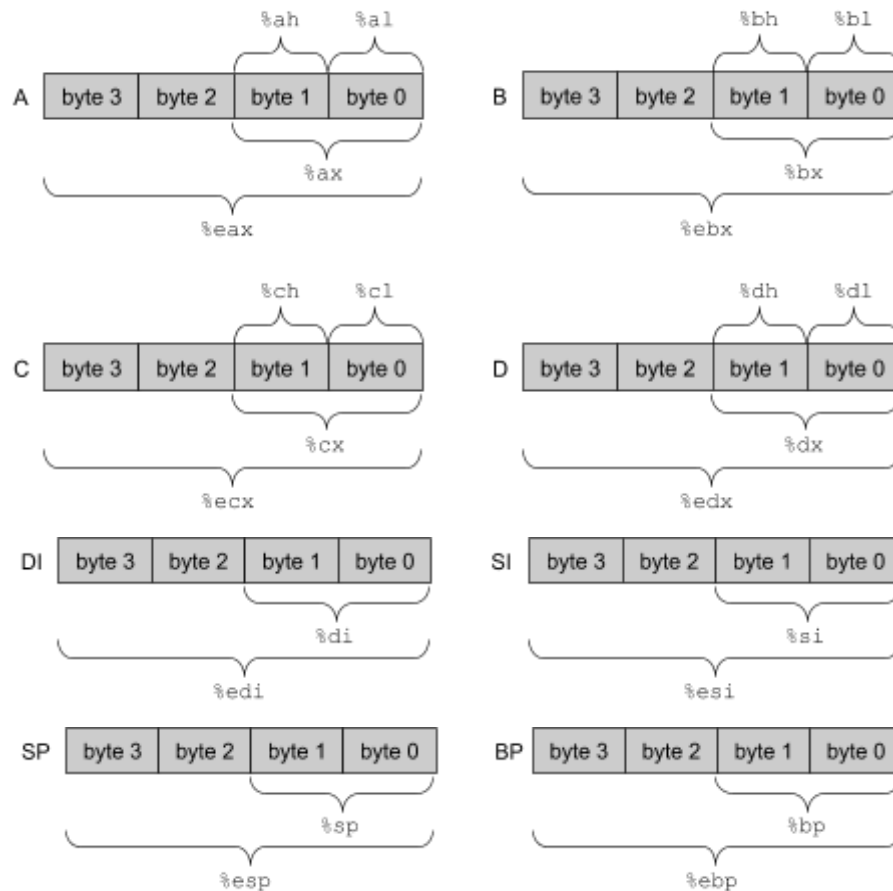
Si noti che il tipo di dato `long long` non è supportato in modo nativo dall'hardware IA32.

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

3.1.3 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) di dimensione 32 bit (4 byte), di cui i primi 6 possono essere usati come se fossero variabili per memorizzare interi e puntatori:



I registri SP e BP hanno invece un uso particolare che vedremo in seguito. Nella descrizione, byte₀ denota il byte **meno significativo** del registro e byte₃ quello **più significativo**.

Si noti che è possibile accedere a singole parti di un registro utilizzando dei nomi simbolici. Ad esempio, per il registro A:

- %eax denota i 4 byte di A (byte₃, byte₂, byte₁, byte₀)
- %ax denota i due byte meno significativi di A (byte₁ e byte₀)
- %al denota il byte meno significativo di A (byte₀)
- %ah denota il secondo byte meno significativo di A (byte₁)

Fanno eccezione i registri DI e SI, dove non si può accedere a un singolo byte.

L'IA32 ha anche altri registri:

- EIP: registro a 32 bit che contiene l'indirizzo della prossima istruzione da eseguire (program counter)
- EFLAGS: registro a 32 bit che contiene informazioni sullo stato del processore
- altri registri per calcoli in virgola mobile e vettoriali che non trattiamo in questa dispensa

3.1.4 Operandi e modi di indirizzamento della memoria

Le istruzioni macchina hanno in genere uno o più **operandi** che definiscono i dati su cui operano. In generale, si ha un **operando sorgente** che specifica un valore di ingresso per l'operazione e un **operando destinazione** che identifica dove deve essere immagazzinato il risultato dell'operazione.

Gli operandi sorgente possono essere di tre tipi:

- *Immediato*: operando immagazzinato insieme all'istruzione stessa;
- *Registro*: operando memorizzato in uno degli 8 registri interi;
- *Memoria*: operando memorizzato in memoria.

Gli operandi destinazione possono essere invece di soli due tipi:

- *Registro*: il risultato dell'operazione viene memorizzato in uno degli 8 registri interi;
- *Memoria*: il risultato dell'operazione viene memorizzato in memoria.

Useremo la seguente notazione:

- Se E è il nome di un registro, $R[E]$ denota il contenuto del registro E ;
- Se x è un indirizzo di memoria, $M_b[x]$ denota dell'oggetto di b byte all'indirizzo x (omettiamo il pedice b quando la dimensione è irrilevante ai fini della descrizione).

Si hanno le seguenti 11 possibili forme di operandi. Per gli operandi di tipo memoria, vi sono vari **modi di indirizzamento** che consentono di accedere alla memoria dopo averne calcolato un indirizzo.

Tipo	Sintassi	Valore denotato	Nome convenzionale
Immediato	$\$imm$	imm	Immediato
Registro	E	$R[E]$	Registro
Memoria	imm	$M[imm]$	Assoluto
Memoria	(E_{base})	$M[R[E_{base}]]$	Indiretto
Memoria	$imm(E_{base})$	$M[imm + R[E_{base}]]$	Base e spiazzamento
Memoria	(E_{base}, E_{indice})	$M[R[E_{base}] + R[E_{indice}]]$	Base e indice
Memoria	$imm(E_{base}, E_{indice})$	$M[imm + R[E_{base}] + R[E_{indice}]]$	Base, indice e spiazzamento
Memoria	(E_{indice}, s)	$M[R[E_{indice}] \cdot s]$	Indice e scala
Memoria	$imm(E_{indice}, s)$	$M[imm + R[E_{indice}] \cdot s]$	Indice, scala e spiazzamento
Memoria	$(E_{base}, E_{indice}, s)$	$M[R[E_{base}] + R[E_{indice}] \cdot s]$	Base, indice e scala

Memoria	$\text{imm}(E_{\text{base}}, E_{\text{indice}}, s)$	$M[\text{imm} + R[E_{\text{base}}] + R[E_{\text{indice}}] \cdot s]$	Base, indice, scala e spiazzamento
---------	---	---	------------------------------------

Negli indirizzamenti a memoria con indice scalato, il parametro s può assumere solo uno dei valori: 1, 2, 4, 8. Il parametro immediato imm è un valore intero costante a 32 bit, ad esempio -24 (decimale) oppure 0xAF25CB7E (esadecimale).

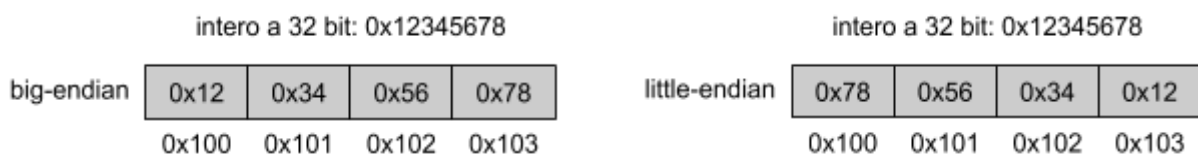
Nel seguito, usiamo la notazione S_n per denotare un operando **sorgente** di n byte, e D_n per denotare un operando **destinazione** di n byte. Omettiamo il pedice quando la dimensione è irrilevante ai fini della descrizione.

3.1.5 Rappresentazione dei numeri in memoria: big-endian vs. little-endian

L'**endianess** di un processore definisce l'**ordine** con cui vengono disposti in **memoria** i **byte** della rappresentazione di un valore numerico:

- **big-endian**: il byte **più** significativo del numero viene posto all'indirizzo più basso;
- **little-endian**: il byte **meno** significativo del numero viene posto all'indirizzo più basso.

Ad esempio, l'intero a 32 bit 0x12345678 viene disposto all'indirizzo 0x100 di memoria con le seguenti sequenze di byte (in esadecimale):



Si noti come nel formato big-endian l'ordine dei byte è lo stesso in cui appare nel letterale numerico che denota il numero, in cui la cifra più significativa appare per prima. Nel little-endian è il contrario.

Esempi di processori big endian sono PowerPC e SPARC. Processori little-endian sono ad esempio quelli della famiglia x86.

3.1.6 Istruzioni di movimento dati

Le istruzioni di movimento dati servono per **copiare byte** da memoria a registro, da registro a registro, e da registro a memoria. Con la notazione $X:Y$ denotiamo la concatenazione delle cifre di X con quelle di Y . Esempio: A3F:C07 denota A3FC07.

3.1.6.1 Stessa dimensione sorgente e destinazione: MOV

Una delle istruzioni più comuni è la MOV, dove sorgente e destinazione hanno la stessa dimensione.

Istruzione	Effetto	Descrizione
<i>MOV S, D</i>	$D \leftarrow S$	<i>copia byte da sorgente S a destinazione D</i>
<i>movb S₁, D₁</i>	$D_1 \leftarrow S_1$	copia 1 byte
<i>movw S₂, D₂</i>	$D_2 \leftarrow S_2$	copia 2 byte
<i>movl S₄, D₄</i>	$D_4 \leftarrow S_4$	copia 4 byte

3.1.6.2 Dimensione destinazione maggiore di quella sorgente: MOVZ, MOVS

Le istruzioni MOVZ, e MOVS servono per spostare dati da un operando sorgente a un operando destinazione di dimensione maggiore. Servono per effettuare le conversioni di tipi interi senza segno (MOVZ) e con segno (MOVS).

Istruzione	Effetto	Descrizione
<i>MOVZ S, D</i>	$D \leftarrow \text{ZeroExtend}(S)$	<i>copia byte da sorgente S a destinazione D, riempiendo con zero i byte che D ha in più rispetto a S</i>
<i>movzbw S₁, D₂</i>	$D_2 \leftarrow 0x00:S_1$	copia 1 byte in 2 byte, estendi con zero
<i>movzbl S₁, D₄</i>	$D_4 \leftarrow 0x000000:S_1$	copia 1 byte in 4 byte, estendi con zero
<i>movzwl S₂, D₄</i>	$D_4 \leftarrow 0x0000:S_2$	copia 2 byte in 4 byte, estendi con zero

Esempi:

Si assuma `%eax=0x12341234` e `%ecx=0xABCDEFAB`:

Istruzione	Risultato (estensione sottolineata)
<code>movzbw %al, %cx</code>	<code>%eax=0x12341234</code> <code>%ecx=0xABCD<u>00</u>34</code>
<code>movzbl %al, %ecx</code>	<code>%eax=0x12341234</code> <code>%ecx=0x<u>000000</u>34</code>
<code>movzwl %ax, %ecx</code>	<code>%eax=0x12341234</code> <code>%ecx=0x<u>0000</u>1234</code>

Vediamo ora l'istruzione MOVS:

Istruzione	Effetto	Descrizione
<i>MOVS S, D</i>	$D \leftarrow \text{SignExtend}(S)$	<i>copia byte da sorgente S a destinazione D, riempiendo con il bit del segno (bit più significativo) di S i byte che D ha in più rispetto a S</i>
<i>movsbw S₁, D₂</i>	$D_2 \leftarrow 0x\underline{MM}:S_1$	copia 1 byte in 2 byte, estendi con <u>M</u> ='F' se il bit più significativo di S ₁ (bit del segno) è 1 e con <u>M</u> ='0' altrimenti
<i>movsbl S₁, D₄</i>	$D_4 \leftarrow 0x\underline{MMMMMM}:S_1$	copia 1 byte in 4 byte, estendi con <u>M</u> ='F' se il bit più significativo di S ₁ (bit del segno) è 1 e con <u>M</u> ='0' altrimenti
<i>movswl S₂, D₄</i>	$D_4 \leftarrow 0x\underline{MMMM}:S_2$	copia 2 byte in 4 byte, estendi con <u>M</u> ='F' se il bit più significativo di S ₂ (bit del segno) è 1 e con <u>M</u> ='0' altrimenti

Esempi:

Si assuma `%eax=0x12341234` e `%ecx=0xABCDE1E2`:

Istruzione	Risultato (estensione sottolineata)
<code>movsbw %al, %cx</code>	<code>%eax=0x123412<u>34</u></code> <code>%ecx=0xABCD<u>00</u>34</code>
<code>movsbl %al, %ecx</code>	<code>%eax=0x123412<u>34</u></code> <code>%ecx=0x<u>000000</u>34</code>
<code>movswl %ax, %ecx</code>	<code>%eax=0x1234<u>1234</u></code> <code>%ecx=0x<u>0000</u>1234</code>
<code>movsbw %cl, %ax</code>	<code>%ecx=0xABCD<u>E1E2</u></code> <code>%eax=0x1234<u>FF</u>E2</code>
<code>movsbl %cl, %eax</code>	<code>%ecx=0xABCD<u>E1E2</u></code> <code>%eax=0x<u>FFFFFF</u>E2</code>
<code>movswl %cx, %eax</code>	<code>%ecx=0xABCD<u>E1E2</u></code> <code>%eax=0x<u>FFFF</u>E1E2</code>

3.1.6.3 Movimento dati da/verso la stack: PUSH, POP

Le istruzioni `PUSH`, e `POP` servono per spostare dati da un operando sorgente verso la cima della stack (`PUSH`) e dalla cima della stack verso un operando destinazione (`POP`):

Istruzione	Effetto	Descrizione
<code>pushl S₄</code>	$R[\%esp] \leftarrow R[\%esp] - 4$ $M[R[\%esp]] \leftarrow S_4$	copia l'operando di 4 byte S sulla cima della stack
<code>popl D₄</code>	$D_4 \leftarrow M[R[\%esp]]$ $R[\%esp] \leftarrow R[\%esp] + 4$	togli i 4 byte dalla cima della stack e copiali nell'operando D

3.1.7 Istruzioni aritmetico-logiche

Le seguenti istruzioni IA32 servono per effettuare operazioni su interi a 1, 2 e 4 byte:

Istruzione	Effetto	Descrizione
INC D	$D \leftarrow D+1$	incrementa destinazione
DEC D	$D \leftarrow D-1$	decrementa destinazione
NEG D	$D \leftarrow -D$	inverte segno destinazione
NOT D	$D \leftarrow \sim D$	complementa a 1 destinazione
ADD S, D	$D \leftarrow D+S$	aggiungi sorgente a destinazione e risultato in destinazione
SUB S, D	$D \leftarrow D-S$	sottrai sorgente da destinazione e risultato in destinazione
IMUL S, D	$D \leftarrow D*S$	moltiplica sorgente con destinazione con segno e risultato in destinazione, la destinazione deve essere un registro
XOR S, D	$D \leftarrow D \wedge S$	or esclusivo sorgente con destinazione e risultato in destinazione
OR S, D	$D \leftarrow D S$	or sorgente con destinazione e risultato in destinazione
AND S, D	$D \leftarrow D \& S$	and sorgente con destinazione e risultato in destinazione

L'istruzione IDIV per effettuare divisioni intere con segno è più complessa, rappresentando uno dei casi in cui operazioni IA32 lavorano su registri specifici, in questo caso D e A.

Istruzione	Effetto	Descrizione
IDIV V	$A \leftarrow D:A / V$ $D \leftarrow D:A \% V$	calcola simultaneamente il quoziente e 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
idivb ⁵ V ₁	$\%al \leftarrow \%ax / V_1$ $\%ah \leftarrow \%ax \% V_1$	calcola simultaneamente il quoziente e il resto della divisione del dividendo %ax con il divisore V ₁ che può essere registro o memoria a 8 bit ⁶ .

⁵ La variante a 8 bit fa eccezione perché il divisore non è ottenuto dalla concatenazione di due altri registri, ma usa direttamente %ax come dividendo.

⁶ Questa versione di divisione a 8 bit fa eccezione alla regola generale che prevede come dividendo la concatenazione di porzioni dei registri D e A.

<code>idivw V₂</code>	$\%ax \leftarrow \%dx:\%ax / V_2$ $\%dx \leftarrow \%dx:\%ax \% V_2$	calcola simultaneamente il quoziente e il resto della divisione del dividendo ottenuto concatenando <code>%dx</code> con <code>%ax</code> , dove <code>%dx</code> ha i bit più significativi, con il divisore V_2 che può essere un registro o memoria a 16 bit
<code>idivl V₄</code>	$\%eax \leftarrow \%edx:\%eax / V_4$ $\%edx \leftarrow \%edx:\%eax \% V_4$	calcola simultaneamente il quoziente e il resto della divisione del dividendo ottenuto concatenando <code>%edx</code> con <code>%eax</code> , dove <code>%dex</code> ha i bit più significativi, con il divisore V_4 che può essere registro o memoria a 32 bit.

Esempio. Si può calcolare ad esempio `%edi = %edi / %esi` come segue:

```

movl %edi, %eax    # carica il dividendo in %eax
movl %eax, %edx    # poiché il dividendo è a 32 bit, basterebbe %eax,
sar $31, %edx      # ma serve comunque replicare il bit più significa-
                  # tivo di %eax in %edx (ci sono altri modi di farlo)
idivl %esi         # divide %edx:%eax per il divisore %esi
movl %eax, %edi    # mette il quoziente %eax calcolato in %edi

```

L'operazione è costosa in termini computazionali e ipotoca i registri A e D, limitando la flessibilità nell'usare i registri nel programma.

3.1.7.1 L'istruzione LEA (load effective address)

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

Istruzione	Effetto	Descrizione
<code>leal S, D₄</code>	$D_4 \leftarrow \&S$	Calcola l'indirizzo effettivo specificato dall'operando di tipo memoria S e lo scrive in D

Si noti 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 però accedere in memoria a quell'indirizzo.

Esempi.

Si assuma `%eax=0x100`, `%ecx=0x7` e $M_4[0x100]=0xCAFE$

Istruzione	Effetto	Risultato
<code>movl (%eax), %edx</code>	$R[\%edx] \leftarrow M_4[R[\%eax]]$	<code>%edx=0xCAFE</code>
<code>leal (%eax), %edx</code>	$R[\%edx] \leftarrow R[\%eax]$	<code>%edx=0x100</code>

Si noti la differenza fra `leal` e `movl` che abbiamo discusso sopra. Si considerino inoltre i seguenti altri esempi:

Istruzione	Effetto	Risultato
<code>leal (%eax, %ecx), %edx</code>	$R[\%edx] \leftarrow R[\%eax] + R[\%ecx]$	<code>%edx=0x107</code>
<code>leal -3(%eax, %ecx), %edx</code>	$R[\%edx] \leftarrow R[\%eax] + R[\%ecx] - 3$	<code>%edx=0x104</code>
<code>leal -3(%eax, %ecx, 2), %edx</code>	$R[\%edx] \leftarrow R[\%eax] + R[\%ecx] \cdot 2 - 3$	<code>%edx=0x10B</code>
<code>leal (%eax, %ecx, 2), %edx</code>	$R[\%edx] \leftarrow R[\%eax] + R[\%ecx] \cdot 2$	<code>%edx=0x10E</code>
<code>leal (%ecx, 4), %edx</code>	$R[\%edx] \leftarrow R[\%ecx] \cdot 4$	<code>%edx=0x1C</code>

L'istruzione `leal` viene usata per scrivere **programmi più veloci** e viene sfruttata tipicamente per due scopi:

1. calcolare l'indirizzo effettivo di un oggetto in memoria una sola volta, per poi usarlo più volte;
2. calcolare **espressioni aritmetiche su interi o puntatori** usando una sola istruzione.

Si noti infatti che, sebbene sia stata pensata per calcolare indirizzi di memoria, la `leal` può essere usata per calcolare espressioni intere che non rappresentano indirizzi.

Esempio.

Si consideri il seguente frammento di programma C:

```
int x=10;
int y=20;
int z=x+y*4-7;
```

Riformuliamo il frammento in modo che ogni operazione aritmetica abbia la forma: $a = a \text{ op } b$, ottenendo il seguente codice equivalente, la corrispondente traduzione in codice IA32 e una versione ottimizzata del codice IA32 basata sull'istruzione `leal`:

Codice C	Codice IA32	Codice IA32 ottimizzato
<pre>int x=10; // x è in %eax int y=20; // y è in %ecx</pre>	<pre>movl \$10, %eax movl \$20, %ecx</pre>	<pre>movl \$10, %eax movl \$20, %ecx</pre>

<pre>int z=y; // z è in %edx z=z*4; z=z-7; z=z+x;</pre>	<pre>movl %ecx,%edx imull \$4,%edx addl \$-7,%edx addl %eax,%edx</pre>	<pre>leal -7(%eax,%ecx,4),%edx</pre>
---	--	--------------------------------------

Si noti che, se l'espressione da calcolare fosse stata $x+y*5-7$, non sarebbe stato possibile usare la `leal`: infatti il fattore moltiplicativo nei vari modi di indirizzamento a memoria (scala) può essere solo 1, 2, 4, 8. Non tutte le espressioni aritmetiche possono quindi essere calcolate con la `leal`.

3.1.7.2 Istruzioni di shift SHL, SHR, SAL, SAR

Le istruzioni di shift consentono di scorrere 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). Uno shift a destra di una posizione corrisponde a una divisione per due dell'operando destinazione, mentre uno shift a sinistra di una posizione corrisponde a una moltiplicazione per due dell'operando destinazione.

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 dove: <ul style="list-style-type: none"> D è registro o memoria (da 1,2,4 byte) X può essere %cl, imm, o 1 se X=1, X può essere omissso
SHR X, D	$D \leftarrow D \gg X$	effettua shift logico a destra dell'operando D di X posizioni dove: <ul style="list-style-type: none"> D è registro o memoria (da 1,2,4 byte) X può essere %cl, imm, o 1 se X=1, X può essere omissso
SAR X, D	$D \leftarrow D \gg X$	effettua shift aritmetico a destra dell'operando D di X posizioni dove: <ul style="list-style-type: none"> D è registro o memoria (da 1,2,4 byte) X può essere %cl, imm, o 1 se X=1, X può essere omissso

Si noti che gli shift a sinistra non influenzano il bit del segno in complemento a due, poiché vengono espulsi verso sinistra, per cui di fatto SHL e SAL hanno la stessa semantica. Infatti gli **opcode** delle loro varie varianti sono **identici**. In pratica sono sinonimi per l'assemblatore. Uno dei loro scopi principali è quello di **moltiplicare** in modo più efficiente di IMUL un operando registro o memoria per una potenza di due.

Consideriamo invece uno shift a destra: nel caso aritmetico il bit del segno deve essere propagato verso destra per preservare il segno dell'operando.

Esempio 1: vediamo uno shift a destra di 3 posizioni, che corrisponde a una divisione per $2^3=8$ dell'operando `%al` assunto con segno:

Prima	Operando	Esadecimale	Binario	Decimale
<code>sarb \$3, %al</code>	<code>%al</code>	CA	1 1001010	-54
Dopo		F9	111 11001	-6 = -54/8

Si noti come lo shift aritmetico inietti da sinistra tre volte 1, che è il bit più significativo (che in complemento a 2 ci dice che il numero è negativo).

Esempio 2: consideriamo ora il medesimo caso in cui però l'operando `%al` è non negativo:

Prima	Operando	Esadecimale	Binario	Decimale
<code>sarb \$3, %al</code>	<code>%al</code>	7A	0 1111010	122
Dopo		0F	000 01111	15 = 122/8

In entrambi i casi la divisione ottenuta è corretta.

Esempio 3: osserviamo infine che uno shift logico a destra inietta sempre uno zero a sinistra, indipendentemente dal segno del bit più significativo dell'operando. Pertanto non dovrebbe mai essere usato se l'intento è dividere per una potenza di 2 un operando che potrebbe essere negativo considerando quindi lo shift come un'operazione di divisione.

Prima	Operando	Esadecimale	Binario	Decimale
<code>shrb \$3, %al</code>	<code>%al</code>	CA	1 1001010	-54
Dopo		19	000 11001	25 <> -54/8

Si noti l'errore di aver usato un'istruzione di shift logico a destra per dividere un numero negativo. L'uso degli operatori di shift logico è particolarmente adatto invece per manipolazioni che richiedono di testare o estrarre porzioni di bit da un operando.

3.1.8 Istruzioni di salto

Normalmente, il flusso del controllo di un programma procede in modo sequenziale, eseguendo le istruzioni nell'ordine in cui appaiono in memoria. Ogni volta che un'istruzione *I* viene eseguita, il registro EIP (instruction pointer), che punta alla prossima istruzione da eseguire, viene incrementato automaticamente del numero di byte occupati dall'istruzione *I*.

Vi sono tuttavia istruzioni, chiamate **istruzioni di salto**, che permettono di alterare il flusso del controllo, modificando il contenuto del registro EIP in modo che l'esecuzione non prosegua con istruzione successiva, ma con un'altra che inizia ad un indirizzo diverso.

Vi sono tre tipi di istruzioni di salto:

1. salti **incondizionati**: il registro EIP viene sovrascritto con l'indirizzo di memoria dell'istruzione a cui si vuole saltare;
2. salti **condizionati**: il registro EIP viene sovrascritto con l'indirizzo di memoria dell'istruzione a cui si vuole saltare, ma solo se è verificata una determinata condizione sui dati;
3. **chiamata e ritorno** da funzione (che vedremo in seguito).

3.1.8.1 Salti incondizionati: JMP

Le istruzioni di salto incondizionato possono essere di tipo diretto o indiretto:

Istruzione	Effetto	Nota
<code>jmp etichetta</code>	$R[\%eip] \leftarrow \text{indirizzo associato all'etichetta}$	salto diretto
<code>jmp *S</code>	$R[\%eip] \leftarrow S$	salto indiretto

Esempio.

Si consideri il seguente frammento di programma x86:

```
    movl $0, %eax
L:  incl %eax
    jmp L
```

Il programma esegue dapprima l'istruzione `movl`, poi `incl`. Quando incontra la `jmp` ritorna ad eseguire la `incl`. Infatti l'etichetta `L` (introdotta con la sintassi `L:`) denota l'indirizzo dell'istruzione `incl`. Si ha quindi un ciclo infinito.

Esempio.

Si consideri il seguente frammento di programma x86:

```
jmp *(%eax)
```

Il programma salta all'indirizzo effettivo denotato dall'operando `(%eax)`. L'operazione effettuata è quindi: $\%eip \leftarrow M[R[\%eax]]$.

3.1.8.2 Salti condizionati e condition code: Jcc, CMP

Le istruzioni di salto condizionato consentono di modificare il registro EIP, e quindi alterare il normale flusso sequenziale del controllo dell'esecuzione, solo se una determinata condizione è soddisfatta. Il test viene effettuato in base al contenuto di un registro particolare chiamato registro dei FLAG, che viene modificato come effetto collaterale dell'esecuzione della maggior parte delle istruzioni aritmetico-logiche.

Un salto condizionato avviene in due passi:

1. un'operazione aritmetico-logica effettua un'operazione sui dati
2. in base all'esito dell'operazione, l'istruzione di salto condizionato salta o meno a un'etichetta

Il registro dei FLAG contiene in particolare quattro codici di condizione (condition code) booleani:

1. **ZF** (zero flag): viene posto a 1 se l'ultima operazione aritmetico-logica ha prodotto un valore zero e 0 se ha prodotto un valore diverso da zero;
2. **SF** (sign flag): viene posto a 1 se l'ultima operazione aritmetico-logica ha prodotto un valore negativo e 0 se ha prodotto un valore non negativo;
3. **CF** (carry flag): viene posto a 1 se l'ultima operazione aritmetico-logica ha generato un riporto e 0 altrimenti;
4. **OF** (overflow flag): viene posto a 1 se l'ultima operazione aritmetico-logica ha generato un overflow e 0 altrimenti.

La forma generale di una istruzione di salto condizionato è la seguente:

Istruzione	Effetto	Nota
Jcc etichetta	if (condizione) R[%eip] ← indirizzo associato all'etichetta	salto condizionato se la condizione associata al suffisso cc è verificata

La seguente tabella riporta le possibili condizioni su cui è possibile saltare e i possibili codici suffissi di istruzione:

Suffisso cc	Sinonimo	Condizione ⁷	Significato
e	z	ZF	Uguale (o zero)
ne	nz	~ZF	Diverso (o non zero)
s		SF	Negativo

⁷ Si ricordi che: ~ denota la negazione logica, & l'and, | l'or, e ^ l'or esclusivo (xor). La condizione in funzione dei condition code ZF, SF, CF, OF viene riportata per completezza e non è in generale utile per il programmatore.

ns		$\sim SF$	Non negativo
g	nle	$\sim (SF \wedge OF) \ \& \sim ZF$	Maggiore (g=greater) con segno
ge	nl	$\sim (SF \wedge OF)$	Maggiore o uguale (ge=greater or equal) con segno
l	nge	$SF \wedge OF$	Minore (l=less) con segno
le	ng	$(SF \wedge OF) \mid ZF$	Minore o uguale con segno
a	nbe	$\sim CF \ \& \sim ZF$	Maggiore (a=above) senza segno
ae	nb	$\sim CF$	Maggiore o uguale (ae=above or equal) senza segno
b	nae	CF	Minore (b=below) senza segno
be	na	$CF \mid ZF$	Minore o uguale (be=below or equal) senza segno

Si noti che i confronti maggiore/minore sono diversi a seconda che si intenda considerare o meno il segno dei valori confrontati.

Esempio 1.

Si consideri il seguente frammento di programma x86 e la sua corrispondente versione C in cui il registro `eax` è trattato come se fosse una variabile:

<code>decl %eax</code> <code>jz L</code>	<code>eax--;</code> <code>if (eax == 0) goto L;</code>
---	---

La prima operazione decrementa il contenuto del registro `eax`. Se `eax` diventa zero, allora l'istruzione `jz` salterà all'etichetta `L`.

Esempio 2.

Si consideri il seguente frammento di programma x86 e la sua corrispondente versione C in cui i registri sono trattati come se fossero variabili:

<code>subl %ebx, %eax</code> <code>je L</code>	<code>temp = eax</code> <code>eax = eax - ebx</code> <code>if (temp == ebx) goto L</code>
---	---

La prima operazione calcola $R[\%eax] - R[\%ebx]$ e scrive il risultato in $R[\%eax]$. Si noti che il risultato della sottrazione è zero se e solo se i due registri sono uguali. Pertanto, l'istruzione `je` salterà all'etichetta `L` se e solo se i due registri sono uguali prima della `SUB`.

Osserviamo che per effettuare un salto condizionato rispetto al contenuto di due registri abbiamo dovuto modificarne uno: infatti la `SUB` modifica l'operando destinazione. Per ovviare a questo problema il set IA32 prevede una istruzione di sottrazione che non modifica l'operando

destinazione, pensata specificamente per essere usata nei confronti:

Istruzione	Effetto	Nota
CMP S,D	calcola D-S	la differenza calcolata viene usata per modificare i condition code e poi va persa

La seguente tabella riporta la condizione testata per ciascun prefisso assumendo di aver appena effettuato una operazione CMP S,D:

Suffisso cc	Sinonimo	Condizione testata dopo istruzione CMP S,D	Ovvero
e	z	D-S == 0	D == S
ne	nz	D-S != 0	D != S
g	nle	D-S > 0	D > S
ge	nl	D-S >= 0	D >= S
l	nge	D-S < 0	D < S
le	ng	D-S <= 0	D <= S
a	nbe	(unsigned) D - (unsigned) S > 0	(unsigned) D > (unsigned) S
ae	nb	(unsigned) D - (unsigned) S >= 0	(unsigned) D >= (unsigned) S
b	nae	(unsigned) D - (unsigned) S < 0	(unsigned) D < (unsigned) S
be	na	(unsigned) D - (unsigned) S <= 0	(unsigned) D <= (unsigned) S

Esempio 3.

Si consideri il seguente frammento di programma x86 e la sua corrispondente versione C in cui i registri sono trattati come se fossero variabili:

<pre> cml %ebx, %eax jle L </pre>	<pre> if (eax <= ebx) goto L; </pre>
-----------------------------------	---

La prima operazione calcola la differenza $R[\%eax] - R[\%ebx]$. La seconda salta se $R[\%eax] - R[\%ebx] \leq 0$.

3.1.8.3 Chiamata e ritorno da funzione: CALL e RET

Un ulteriore tipo di istruzione di salto è quello relativo alle chiamate e ritorno da funzione:

Istruzione	Effetto	Nota
CALL S	$R[\%esp] \leftarrow R[\%esp] - 4$ $M[R[\%esp]] \leftarrow R[\%eip]$ $R[\%eip] \leftarrow S$	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	$R[\%eip] \leftarrow M[R[\%esp]]$ $R[\%esp] \leftarrow R[\%esp] + 4$	Ritorno da funzione: toglie dalla stack l'indirizzo di ritorno e lo scrive in EIP

Esempio.

Si consideri il seguente frammento di programma x86 e la sua corrispondente versione C in cui i registri sono trattati come se fossero variabili:

<pre> call f imull \$3, %eax ... f: movl \$2, %eax ret </pre>	<pre> f(); eax = eax*3; ... void f() { eax = 2; } </pre>
---	--

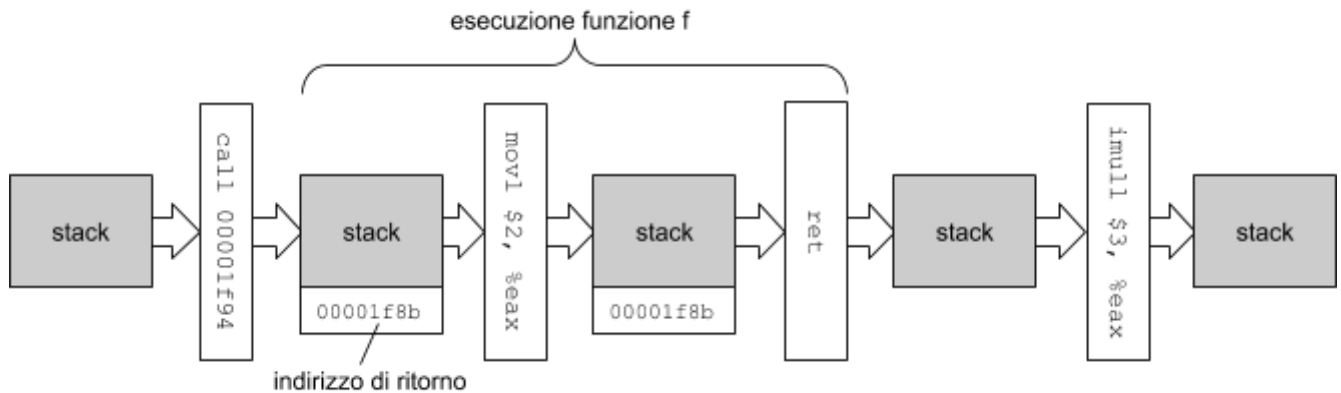
Immaginiamo che il programma sia disposto in memoria ai seguenti indirizzi:

00001f86	call	00001f94	; chiama f
00001f8b	imull	\$3, %eax	
00001f8e	...		
00001f94	movl	\$2, %eax	
00001f99	ret		

Eseguendo le istruzioni a partire dall'indirizzo 00001f86, il flusso delle istruzioni e il loro effetto sui principali registri usati è il seguente:

	%eip (prima)	%eax (prima)	istruzione eseguita	%eip (dopo)	%eax dopo
1	00001f86	-	call 00001f94	00001f94	-
2	00001f94	-	movl \$2, %eax	00001f99	00000002
3	00001f99	00000002	ret	00001f8b	00000002
4	00001f8b	00000002	imull \$3, %eax	00001f8e	00000006

Analizziamo ora il contenuto della stack prima e dopo ogni istruzione:



Si noti che la `CALL` deposita in stack l'indirizzo dell'istruzione successiva, in modo che la `RET` possa proseguire da quella istruzione una volta terminata la chiamata della funzione.

3.1.9 Altre istruzioni e vincoli sulle istruzioni

3.1.9.1 Istruzioni di assegnamento condizionato: `CMOVcc`

L'istruzione `CMOVcc` consente di effettuare degli assegnamenti solo se una determinata condizione è verificata. L'istruzione si basa sulle medesime condizioni della `Jcc`, salvo che invece di saltare, copia l'operando sorgente in quello destinazione.

Istruzione	Effetto	Nota
<code>CMOVcc S,D</code>	<code>if (condizione) D ← S</code>	se la condizione associata al suffisso <code>cc</code> è verificata, copia la sorgente nella destinazione

L'istruzione semplifica alcune operazioni condizionali riducendo il numero di istruzioni richieste. Diversamente dalla `MOV`, l'operando **sorgente** di una `CMOVcc` **non può essere un operando immediato**, la **destinazione deve essere un registro** e **solo operandi a 16 e 32 bit** sono supportati.

Esempio.

Si consideri il seguente frammento di programma x86 e la sua corrispondente versione C in cui i registri sono trattati come se fossero variabili:

<pre> cmpl %ecx, %eax cmovgl %eax, %ecx </pre>	<pre> if (eax > ecx) ecx = eax; </pre>
--	---

La prima istruzione calcola $R[\%eax] - R[\%ecx]$. La seconda sovrascrive $R[\%ecx]$ con $R[\%eax]$ se $R[\%eax] > R[\%ecx]$.

3.1.9.2 Altre istruzioni di confronto: TEST

Nello stesso spirito della `CMP`, che corrisponde a una `SUB` in cui non viene modificato l'operando destinazione, l'istruzione `TEST` è identica a una `AND`, tranne che non modifica l'operando destinazione:

Istruzione	Effetto	Nota
<code>TEST S,D</code>	calcola <code>S & D</code>	l'and bit a bit fra gli operandi calcolato viene usato per modificare i condition code e poi va perso

Esempio.

L'istruzione `TEST` può essere usata al posto della `CMP` ad esempio per verificare se un registro è zero o meno:

<code>testl %eax, %eax</code> <code>jz L</code>	<code>cmpl \$0, %eax</code> <code>jz L</code>	<code>if (eax==0) goto L</code>
--	--	---------------------------------

Si noti che l'AND di un valore con se stesso è zero se e solo se il valore è zero.

3.1.9.3 Altre istruzioni di assegnamento: SETcc

L'istruzione `SETcc` permette di assegnare i valori 0 o 1 a un registro a 8 bit o a un byte di memoria, a seconda che una data condizione sul registro `EFLAGS` sia verificata:

Istruzione	Effetto	Nota
<code>SETcc D₁</code>	$D_1 \leftarrow \text{condizione}$	se la condizione associata al suffisso <code>cc</code> è verificata, scrive 1 in <code>D₁</code> , altrimenti scrive 0

Esempio.

L'istruzione `SETcc` può essere usata al posto della `Jcc` ad esempio per calcolare il valore di un'espressione booleana:

<code>cmpl \$0, %eax</code> <code>setl %dl</code>	<code>dl = (eax < 0)</code>
--	--------------------------------

3.1.9.4 Vincoli sulle istruzioni

Alcune istruzioni hanno dei vincoli sugli operandi che possono prendere. Elenchiamo i vincoli più comuni:

Istruzione	Vincolo	Esempio
IMUL	la destinazione deve essere un registro	imull \$2, %eax # ok imull \$2, (%edi) # errore
CMOVcc	la sorgente non può essere un immediato	cmovgel %ecx, %eax # ok cmovgel \$2, %eax # errore
	la destinazione deve essere un registro	cmovgel %eax, %eax # ok cmovgel %eax, (%edi) # errore
	solo operandi 16 o 32 bit	cmovgeb %cl, %al # errore
TEST e CMP	secondo operando ("destinazione") non può essere immediato	testl %eax, %ecx # ok testl (%eax), \$2 # errore
MOVZ e MOVS	sorgente non immediato e destinazione solo registro	movzbl (%eax), %ecx # ok movzbl \$2, %ecx # errore movzbl %eax, (%ecx) # errore
IDV	usa solo i registri D ed A e l'operando dividendo non può essere immediato	movl %edi # ok movl %1 # errore

Inoltre, si noti che **in generale non è possibile avere due operandi memoria**. Ad esempio: `movl (%eax), (%ecx)` non è consentito.

3.2 Traduzione dei costrutti C in assembly IA32

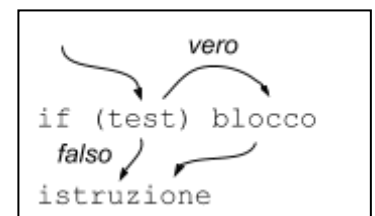
In questo paragrafo vediamo come i compilatori moderni come `gcc` traducono i costrutti del linguaggio C in codice IA32. Si noti come lo stesso programma C potrebbe essere tradotto in assembly in tanti modi diversi. Versioni diverse del compilatore oppure livelli di ottimizzazione diversi portano a codice assembly diverso. Per indicare la traduzione di un frammento di codice `x` in assembly IA32, useremo la notazione `IA32(x)`.

3.2.1 Istruzioni condizionali

Le istruzioni ed espressioni condizionali vengono normalmente basate sulle istruzioni di salto condizionato. In alcuni casi è possibile usare l'istruzione di movimento dati condizionale (`CMOV`).

3.2.1.1 Istruzione `if`

Consideriamo lo schema generale di una istruzione `if`. Se il test effettuato è vero, viene eseguito il blocco e si riprende dall'istruzione successiva, altrimenti si prosegue direttamente con l'istruzione successiva.



L'istruzione `if` può essere tradotta come segue:

C da tradurre	C equivalente	Traduzione IA32
<pre>if (test) blocco istruzione</pre>	<pre>if (!test) goto L; blocco; L: istruzione;</pre>	<pre>IA32(test) Jcc L IA32(blocco) L: IA32(istruzione)</pre>

Si noti che l'`if` viene realizzato effettuando un salto che evita di eseguire il blocco dell'`if` se il test non è soddisfatto. Si salta quindi su `!test` e non su `test`.

Esempio.

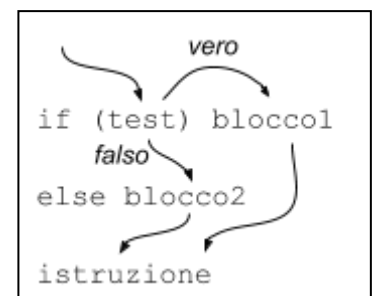
Consideriamo il seguente frammento di programma C con variabili intere senza segno e assumiamo che la variabile `a` sia tenuta nel registro `eax`, `b` in `ebx` e `c` in `ecx`:

C da tradurre	C equivalente	Traduzione IA32
<pre>if (a>b) c=10; c++;</pre>	<pre>if (a<=b) goto L; c=10; L: c++;</pre>	<pre>cmpl %ebx, %eax jbe L movl \$10, %ecx L: incl %ecx</pre>

Notiamo che il test `a<=b` su variabili senza segno viene realizzato calcolando la differenza $R[\text{eax}] - R[\text{ebx}]$ con la `CMP` e saltando se il risultato è ≤ 0 (suffisso `be`=below or equal, confronto senza segno).

3.2.1.1 Istruzione `if...else`

Consideriamo lo schema generale di una istruzione `if...else`. Se il test effettuato è vero, viene eseguito il blocco 1 e si riprende dall'istruzione successiva all'`if...else`, altrimenti si esegue il blocco 2 e si riprende dall'istruzione successiva all'`if...else`.



L'istruzione `if...else` può essere tradotta come segue:

C da tradurre	C equivalente ⁸	Traduzione IA32
<pre>if (test) blocco1 else blocco2 istruzione</pre>	<pre>if (!test) goto E; blocco1 goto F; E: blocco2 F: istruzione;</pre>	<pre>IA32(test) Jcc E IA32(blocco1) jmp F E: IA32(blocco2) F: IA32(istruzione)</pre>

⁸ Si noti che in C il costrutto `if...else` può essere riscritto in termini di `if` e `goto`.

Si noti che l'`if...else` viene realizzato effettuando un salto al blocco 2 che evita di eseguire il blocco 1 se il test non è soddisfatto. Alla fine del blocco 1 c'è un salto incondizionato che evita di eseguire il blocco 2 se il blocco 1 è stato eseguito. Questo realizza la mutua esclusione tra i blocchi eseguiti.

Esempio.

Consideriamo il seguente frammento di programma C con variabili intere con segno e assumiamo che la variabile `a` sia tenuta nel registro `eax`, `b` in `ebx` e `c` in `ecx`:

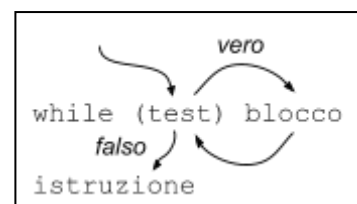
C da tradurre	C equivalente	Traduzione IA32
<pre>if (a<=b) c=10; else c=20; c++;</pre>	<pre>if (a>b) goto E; c=10; goto F; E: c=20; F: c++;</pre>	<pre>cmpl %ebx, %eax jg L movl \$10, %ecx jmp F E: movl \$20, %ecx F: incl %ecx</pre>

Notiamo che il test `a>b` su variabili con segno viene realizzato calcolando la differenza `R[%eax]-R[%ebx]` con la `CMP` e saltando se il risultato è `>0` (suffisso `g`=greater, confronto con segno).

3.2.2 Cicli

3.2.2.1 Istruzione `while`

Consideriamo lo schema generale di una istruzione `while`. Se il test effettuato è vero, viene eseguito il blocco e si ritorna al test, altrimenti si prosegue con l'istruzione successiva al `while`.



L'istruzione `while` può essere tradotta come segue:

C da tradurre	C equivalente ⁹	Traduzione IA32
<pre>while (test) blocco istruzione</pre>	<pre>L: if (!test) goto E; blocco; goto L; E: istruzione;</pre>	<pre>L: IA32(test) Jcc E IA32(blocco) jmp L E: IA32(istruzione)</pre>

Si noti che il `while` è del tutto simile all'`if`, tranne che dopo l'esecuzione del blocco non si prosegue all'istruzione successiva, ma si torna al test.

Esempio.

⁹ Si noti che in C il costrutto `while` può essere riscritto in termini di `if` e `goto`.

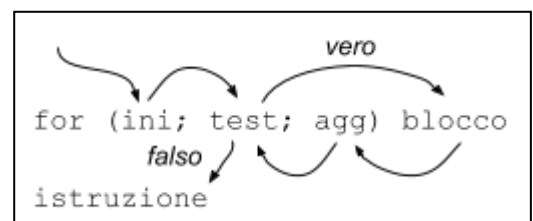
Consideriamo il seguente frammento di programma C con variabili intere senza segno e assumiamo che la variabile *a* sia tenuta nel registro *eax*, *b* in *ebx* e *c* in *ecx*:

C da tradurre	C equivalente	Traduzione IA32
<pre>a=1; c=0; while (a<=b) { c+=a; a++; }</pre>	<pre>a=1; c=0; L: if (a>b) goto E; c+=a; a++; goto L; E:</pre>	<pre>movl \$1, %eax movl \$0, %ecx L: cmpl %ebx, %eax ja E addl %eax, %ecx incl %eax jmp L E:</pre>

Il programma calcola in *c* la somma dei primi *b* interi, cioè $c \leftarrow 1+2+3+\dots+b$.

3.2.2.1 Istruzione `for`

Consideriamo lo schema generale di una istruzione `for`. Si esegue dapprima l'inizializzazione e poi si effettua il test. Se il test effettuato è vero, viene eseguito il blocco, si esegue l'aggiornamento, e si ritorna al test, altrimenti si prosegue con l'istruzione successiva al `for`.



L'istruzione `for` può essere tradotta come segue:

C da tradurre	C equivalente ¹⁰	Traduzione IA32
<pre>for (ini;test;agg) blocco istruzione</pre>	<pre>ini; L: if (!test) goto E; blocco; agg; goto L; E: istruzione</pre>	<pre>IA32(ini) L: IA32(test) Jcc E; IA32(blocco) IA32(agg) jmp L E: IA32(istruzione)</pre>

Esempio.

Consideriamo il seguente frammento di programma C con variabili intere senza segno e assumiamo che la variabile *a* sia tenuta nel registro *eax*, *b* in *ebx* e *c* in *ecx*:

C da tradurre	C equivalente	Traduzione IA32
<pre>c=0;</pre>	<pre>c=0; a=1;</pre>	<pre>movl \$0, %ecx movl \$1, %eax</pre>

¹⁰ Si noti che in C il costrutto `for` può essere riscritto in termini di `if` e `goto`.

<pre>for (a=1; a<=b; a++) c+=a;</pre>	<pre>L: if (a>b) goto E; c+=a; a++; goto L; E:</pre>	<pre>L: cmpl %ebx, %eax ja E addl %eax, %ecx incl %eax jmp L E:</pre>
--	---	---

Il programma calcola in c la somma dei primi b interi, cioè $c \leftarrow 1+2+3+\dots+b$, ed è del tutto equivalente a quello visto come esempio per il `while`.

3.2.3 Funzioni

Una funzione C è normalmente tradotta in assembly IA32 come una sequenza di istruzioni terminate da una `RET` e viene invocata mediante l'istruzione `CALL`. Durante una chiamata a funzione, la funzione che ha effettuato l'invocazione viene detta **chiamante** (caller) e quella invocata viene detta **chiamato** (callee).

Le **convenzioni** relative alla traduzione delle funzioni, del passaggio dei parametri e della restituzione dei valori che vedremo in questo paragrafo non sono specificate dall'ISA IA32, ma sono conformi con la [System V Application Binary Interface](#) (ABI), che descrive uno standard diffuso (es. Mac OS X e Linux) usato nella creazione dei file oggetto e nell'orchestrazione dell'esecuzione dei programmi su piattaforme IA32.

Esempio.

Il seguente frammento di programma C mostra come la definizione di una funzione e la chiamata a funzione vengono tradotte in codice IA32:

C da tradurre	Traduzione IA32
<pre>void f(){ g(); h(); } void g(){ ... } void h(){ ... }</pre>	<pre>f: call g call h ret g: ... ret h: ... ret</pre>

3.2.3.1 Restituzione valore

Per convenzione, valori scalari come interi e puntatori¹¹ vengono restituiti al chiamante dal chiamato scrivendoli nel registro `eax`.

Esempio.

Consideriamo il seguente frammento di programma C:

C da tradurre	C equivalente	Traduzione IA32
<pre>int f(){ return 7+g(); } int g(){ return 10; }</pre>	<pre>int f(){ int tmp = g(); tmp += 7; return tmp; } int g(){ tmp = 10; return tmp; }</pre>	<pre>f: call g addl \$7, %eax ret g: movl \$10, %eax ret</pre>

¹¹ Non trattiamo il caso di come vengono restituiti valori in virgola mobile e strutture. Per approfondimenti si veda ad esempio la documentazione Apple su [IA-32 Function Calling Conventions](#).

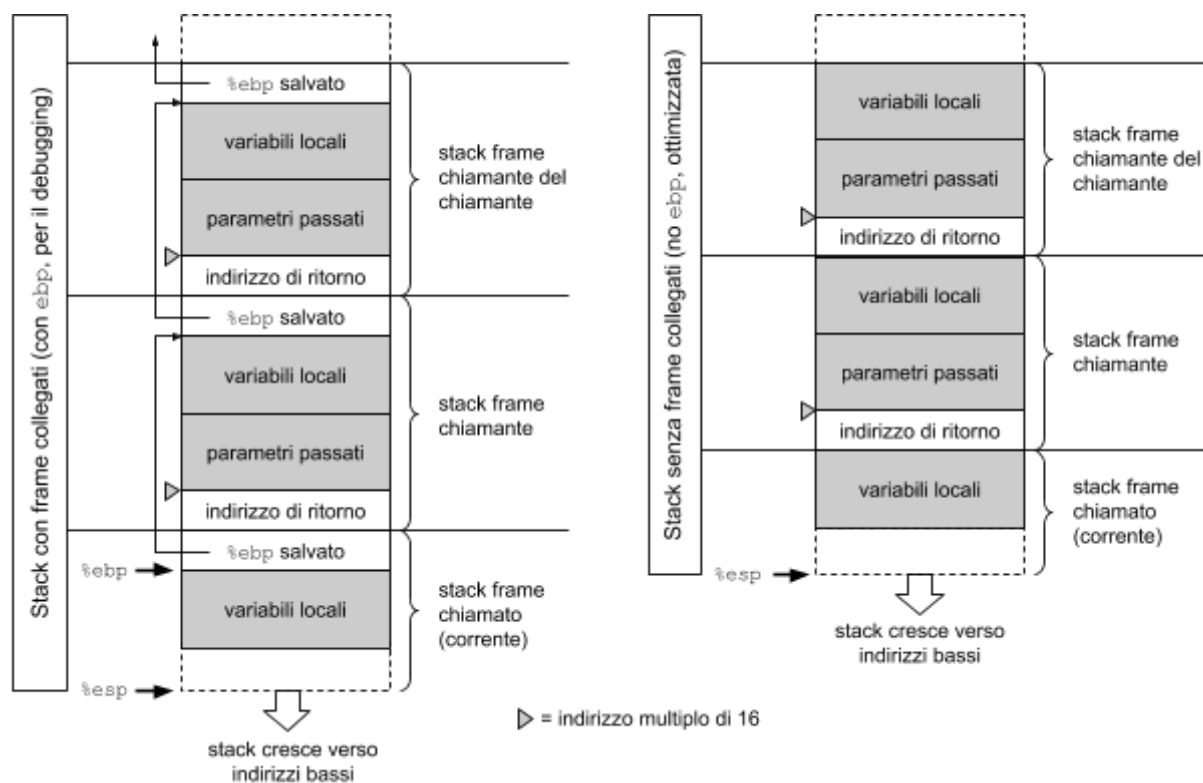
Si noti che `g` restituisce a `f` il valore 10 in `eax`, e a sua volta `f` restituisce al proprio chiamante il valore 17 in `eax`.

3.2.3.2 Stack frame e registro EBP

La stack è uno strumento essenziale per l'orchestrazione delle chiamate a funzione e per fornire spazio di memorizzazione locale alle chiamate. Ogni invocazione a funzione ha associato uno **stack frame** (o record di attivazione), che contiene spazio per memorizzare variabili locali, parametri passati ad altre funzioni, ecc.

Per consentire a un **debugger** di elencare in ogni istante le funzioni pendenti che portano dal main alla funzione correntemente eseguita, e quindi comprendere meglio il contesto in cui una funzione agisce, gli stack frame vengono organizzati a formare concettualmente una **lista collegata**, in cui il registro `ebp` punta al primo stack frame (quello della funzione correntemente eseguita). Ogni stack frame conterrà un puntatore allo stack frame del proprio chiamante.

Poiché il **collegamento fra stack frame usando `ebp` è opzionale**, illustriamo di seguito la struttura con cui viene organizzata la stack sia con che senza collegamento tra frame:

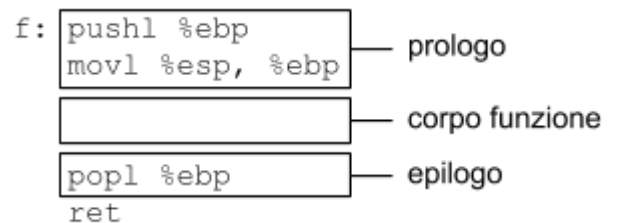


Per convenzione, **nel momento in cui si effettua un'istruzione `CALL`, la base della stack puntata dal registro `%esp` deve essere sempre a un indirizzo multiplo di 16.**¹²

¹² Poiché questa convenzione ha motivazioni prestazionali e non ha implicazioni sulla correttezza di un programma, in alcuni degli esempi in questa dispensa è volutamente ignorata per rendere il codice più semplice da comprendere. Tenere a mente la convenzione è comunque utile per capire perché il codice generato da `gcc` contiene istruzioni apparentemente inutili il cui unico scopo è l'allineamento della stack a multipli di 16.

La lista di stack frame viene gestita mediante un codice di **prologo** all'inizio di una funzione e un codice di **epilogo** alla fine:

- Il **prologo** salva in stack il contenuto di `ebp` (che punta allo stack frame del chiamante) mediante l'istruzione `pushl %ebp`. Il registro base pointer `ebp` viene poi fatto puntare alla posizione corrente in stack contenuta nel registro stack pointer `esp` mediante l'istruzione `movl %esp, %ebp` (così facendo, registro `ebp` viene a puntare allo stack frame corrente).
- L'**epilogo** ripristina il valore di `ebp` che si aveva prima dell'attivazione della funzione corrente eseguendo `popl %ebp`. Il registro `ebp` verrà quindi a puntare nuovamente allo stack frame del chiamante.



In `gcc`, è possibile **omettere il collegamento fra stack frame** compilando con l'opzione `-fomit-frame-pointer`. In questo modo, non verranno generati prologo ed epilogo: la funzione sarà più veloce e compatta, ma il debugging potrebbe essere più difficoltoso.

Esempio.

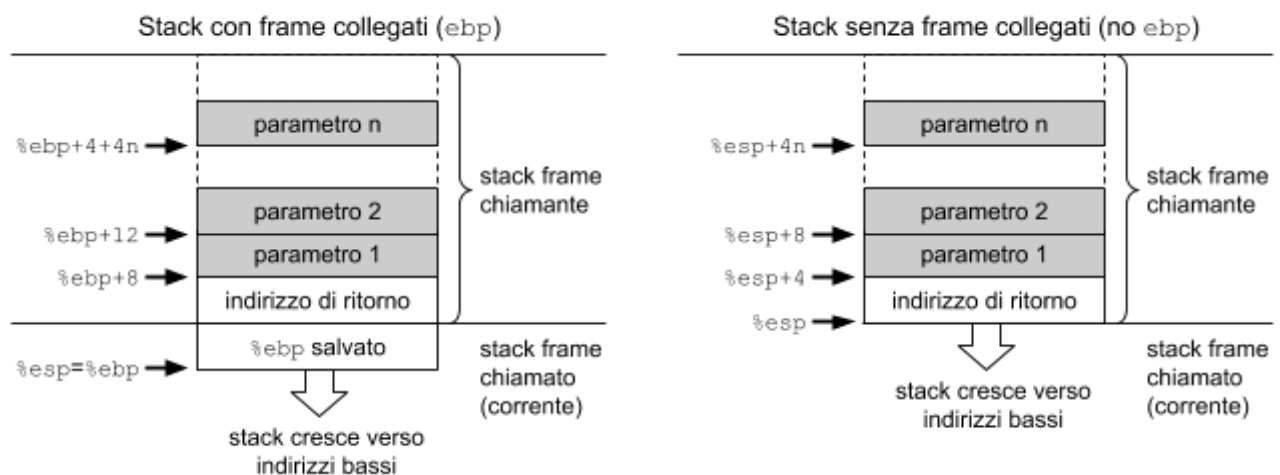
Il seguente esempio mostra come una funzione C viene compilata con e senza l'uso del registro base pointer `ebp`:

C da tradurre <code>test.c</code>	Traduzione IA32 (con <code>ebp</code>): <code>gcc -S test.c</code>	Traduzione IA32 (no <code>ebp</code>): <code>gcc -S -fomit-frame-pointer test.c</code>
<pre>int f() { return 10; }</pre>	<pre>f: pushl %ebp movl %esp, %ebp movl \$10, %eax popl %ebp ret</pre>	<pre>f: movl \$10, %eax ret</pre>

3.2.3.3 Passaggio dei parametri

I parametri di tipi primitivi¹³ vengono passati dal chiamante al chiamato **sulla stack** e vengono disposti in memoria nello stack frame del chiamante **nello stesso ordine** in cui appaiono nell'intestazione della funzione. Parametri interi di 1 o 2 byte vengono **promossi** a 4 byte, in modo che ogni parametro passato sia di dimensione multiplo di 4 byte.

¹³ Non trattiamo il passaggio per parametro di oggetti di tipo struttura. Per approfondimenti si veda ad esempio la documentazione Apple su [IA-32 Function Calling Conventions](https://developer.apple.com/library/ios/qa/qa1064/_index.html).



Esempio.

Il seguente esempio mostra come una funzione C con parametri viene compilata con e senza prologo/epilogo:

C da tradurre	Traduzione IA32 (ebp):	Traduzione IA32 (no ebp):
int f(int x, int y) {	f: pushl %ebp movl %esp, %ebp	f:
return x+y;	movl 8(%ebp), %eax addl 12(%ebp), %eax	movl 4(%esp), %eax addl 8(%esp), %eax
	popl %ebp	
}	ret	ret

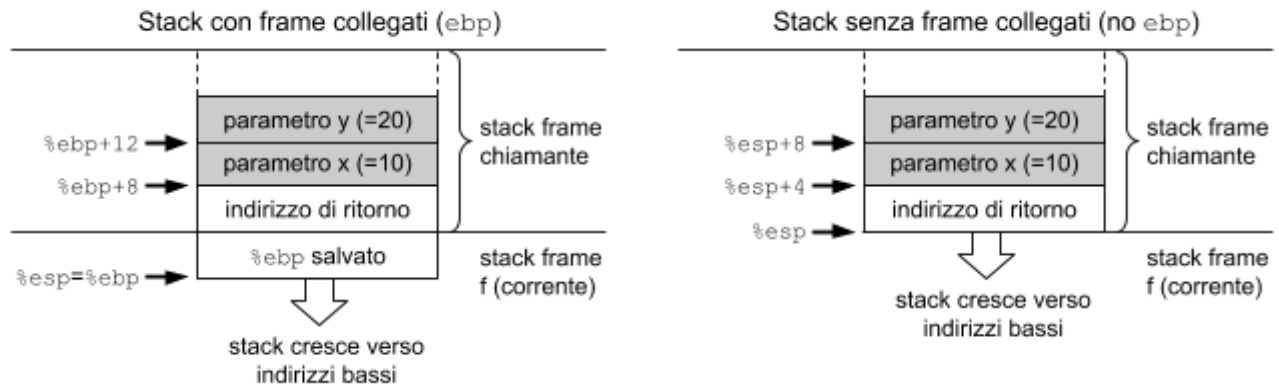
Si noti che il chiamato accede ai parametri passati dal chiamante usando ebp se i frame sono collegati. Se invece i frame non sono collegati, il chiamato accede ai parametri usando esp.

Vediamo ora come la funzione f può essere invocata mostrando il **passaggio dei parametri**. Assumiamo che la variabile locale c sia memorizzata nel registro ecx:

C da tradurre	Traduzione IA32 (no ebp)
c = f(10, 20);	pushl \$20 # passa il secondo parametro pushl \$10 # passa il primo parametro call f # chiama la funzione f addl \$8, %esp # toglie i due parametri dalla stack movl %eax, %ecx # assegna il risultato a c

Si noti che, poiché la stack cresce da indirizzi alti verso indirizzi bassi, le operazioni di push dei parametri su stack avvengono nell'ordine inverso in cui appaiono nella chiamata in modo che

risultino poi disposti in memoria nello stesso ordine. La seguente figura illustra lo stato della stack durante l'esecuzione del corpo della funzione `f`, con e senza frame collegati:



Osserviamo inoltre che i parametri passati sulla stack dal chiamante devono essere poi rimossi dal chiamante stesso dopo la chiamata. Nel nostro esempio, questo si ottiene incrementando lo stack pointer di 8 (`addl $8, %esp`), compensando le due push di 4 byte ciascuna effettuate prima della chiamata (`pushl $20` e `pushl $10`).

3.2.3.4 Registri caller-save e callee-save

L'esecuzione di una funzione potrebbe sovrascrivere i registri in uso al chiamante. Se il chiamante vuole avere la garanzia che il contenuto di un registro non verrà alterato a fronte dell'invocazione di una funzione, è necessario che il suo contenuto venga salvato da qualche parte, generalmente sulla stack. Si hanno due possibilità:

1. Il registro viene **salvato in stack dal chiamante (caller-save)** prima dell'invocazione e ripristinato subito dopo.
2. Il registro viene **salvato in stack dal chiamato (callee-save)** prima di eseguirne il corpo e ripristinato prima di ritornare al chiamato (il salvataggio avviene nel prologo e il ripristino nell'epilogo).

Per convenzione, alcuni registri vengono salvati dal chiamante, e altri dal chiamato:

1. **Registri caller-save:** A, C, D
2. **Registri callee-save:** B, DI, SI, SP, BP

I registri caller-save possono essere liberamente usati da una funzione senza dover essere salvati nel prologo e ripristinati nell'epilogo, ma devono essere salvati/ripristinati a fronte di una chiamata a funzione se serve mantenerne il contenuto dopo la chiamata. I registri callee-save, se usati da una funzione, devono essere salvati nel prologo e ripristinati nell'epilogo della funzione; si ha la garanzia che il loro contenuto sia preservato a fronte dell'invocazione di una funzione.

Esempio (caller-save).

Consideriamo il seguente frammento di programma C con variabili intere e assumiamo che la variabile `a` sia tenuta nel registro `eax` e la variabile `c` in `ecx`:

C da tradurre	C equivalente	Traduzione IA32 (no ebp)
<pre>int f() { return g()+h(); }</pre>	<pre>int f(){ int c = g(); int a = h(); a += c; return a; }</pre>	<pre>f: call g movl %eax, %ecx pushl %ecx call h popl %ecx addl %ecx, %eax ret</pre>

Si noti che il valore restituito da `g` viene scritto in `ecx`, che è un registro caller-save. Se non prendessimo provvedimenti, il suo valore potrebbe essere modificato dalla chiamata ad `h`, perdendo il valore restituito da `g`. Il registro `ecx` viene pertanto salvato in stack (`pushl %ecx`) prima della chiamata ad `h` e ripristinato subito dopo (`popl %ecx`).

Esempio (callee-save).

Vediamo lo stesso esempio di prima in cui usiamo un registro callee-save (`B`) invece che caller-save (`C`) per preservare il valore restituito da `g` a fronte della chiamata ad `h` (assumiamo che la variabile `a` sia tenuta nel registro `eax` e la variabile `b` in `ebx`):

C da tradurre	C equivalente	Traduzione IA32 (no ebp)
<pre>int f() { return g()+h(); }</pre>	<pre>int f(){ int b = g(); int a = h(); a += b; return a; }</pre>	<pre>f: pushl %ebx # prologo call g movl %eax, %ebx call h addl %ebx, %eax popl %ebx # epilogo ret</pre>

Si noti che il contenuto di `ebx` non viene alterato dalla chiamata ad `h` (se infatti `h` dovesse usarlo, dovrebbe salvarlo e poi ripristinarlo prima di terminare), e può quindi essere sommato al valore restituito da `h` (`addl %ebx, %eax`) per determinare il valore restituito da `f`. Il prezzo per usare `ebx` (callee-save) in `f` è che deve essere salvato nel prologo e ripristinato nell'epilogo di `f`.

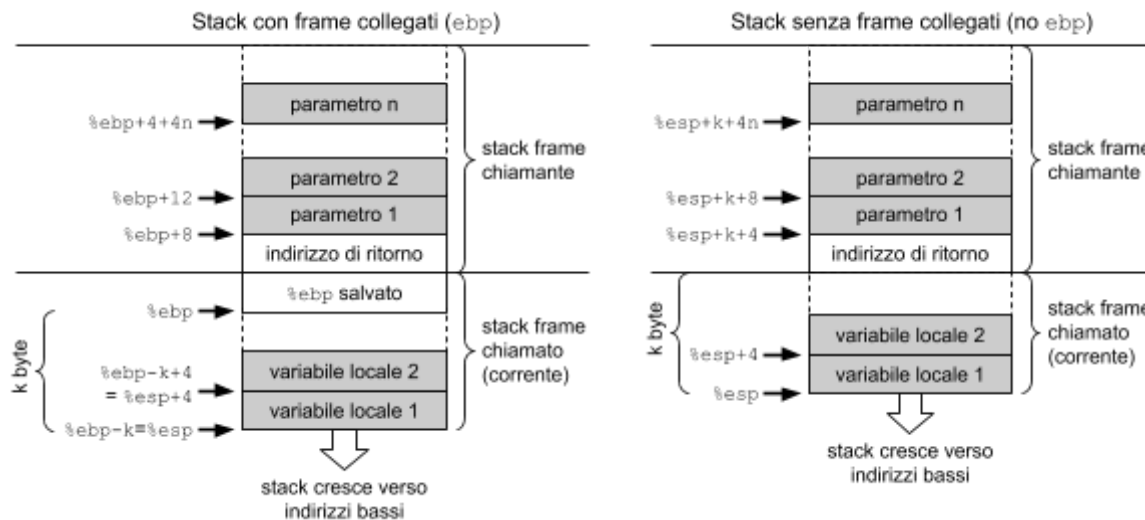
3.2.3.5 Variabili locali

Negli esempi visti finora abbiamo sempre assunto che le variabili locali venissero tenute nei registri. Questa è senz'altro la scelta più semplice e anche la migliore dal punto di vista prestazionale. Tuttavia, alle volte è necessario che le variabili locali abbiano un loro spazio riservato nello stack frame della funzione:

1. se la variabile è di tipo array o struttura e quindi non può essere memorizzata in un registro;
2. se la funzione usa più variabili locali di quanti siano i registri disponibili; oppure

- se la funzione usa l'operatore `&` su una variabile locale, che quindi deve possere un indirizzo in memoria.

Lo spazio per le variabili locali, normalmente allocato in stack nel prologo e deallocato nell'epilogo, è organizzato come segue:



Per accedere a una variabile locale, è possibile usare il registro `esp` con offset positivo. Se il registro `ebp` viene usato per puntare al frame corrente, è possibile usare equivalentemente `ebp` con offset negativo.

Esempio.

Consideriamo il seguente frammento di programma C:

C da tradurre	C equivalente	Traduzione IA32 (con ebp)
<pre>int f(int x){ int y; leggi(&y); return x+y; }</pre>	<pre>int f(int x){ int y; int* c=&y; leggi(c); int a = x; a += y; return a; }</pre>	<pre>f: pushl %ebp # prologo movl %esp, %ebp # prologo subl \$4, %esp # prologo leal -4(%ebp), %ecx # y in -4(%ebp) pushl %ecx # passa param. call leggi addl \$4, %esp # toglie param. movl 8(%ebp), %eax # x in 8(%ebp) addl -4(%ebp), %eax addl \$4, %esp # epilogo popl %ebp # epilogo ret</pre>

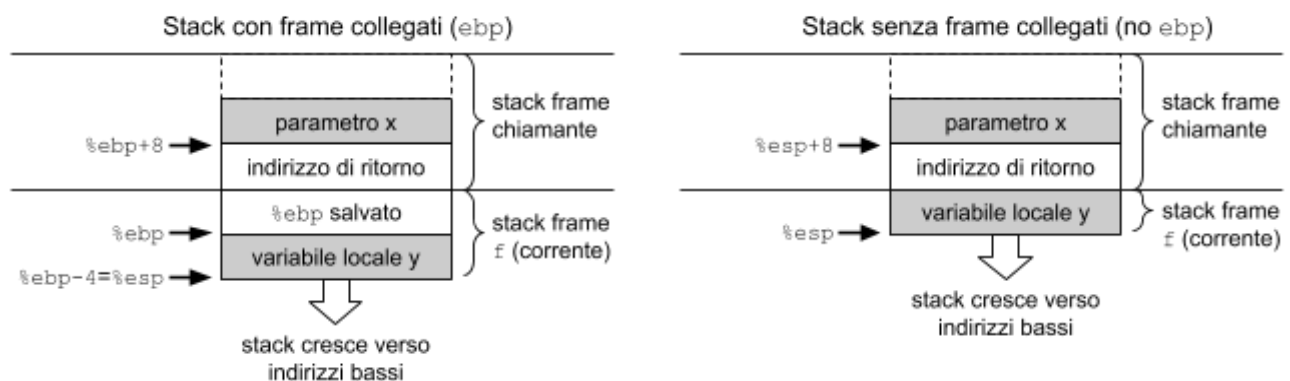
Osserviamo che la zona di memoria per le variabili locali ha dimensione `k=4 byte`. Si noti il modo in cui è compilata l'istruzione `c=&y`: viene usata una `leal` per scrivere nel registro `ecx` l'indirizzo effettivo di `y` (`%ebp-4`). Vediamo ora una versione equivalente del programma senza stack frame

collegati:

C da tradurre	C equivalente	Traduzione IA32 (senza ebp)
<pre>int f(int x){ int y; leggi(&y); return x+y; }</pre>	<pre>int f(int x){ int y; int* c=&y; leggi(c); int a = x; a += y; return a; }</pre>	<pre>f: subl \$4, %esp # prologo leal (%esp), %ecx # y in (%esp) pushl %ecx # passa param. call leggi addl \$4, %esp # toglie param. movl 8(%esp), %eax # x in 8(%esp) addl (%esp), %eax addl \$4, %esp # epilogo ret</pre>

In questo caso si accede al parametro formale x e alla variabile locale y usando lo stack pointer esp piuttosto che il base pointer ebp .

Nella figura seguente illustriamo la struttura (layout) della stack all'inizio dell'esecuzione del corpo della funzione f per entrambe le versioni (con e senza ebp):



3.2.4 Array e aritmetica dei puntatori

L'accesso alle celle di array con elementi di dimensione fino a 4 byte avviene normalmente sfruttando gli indirizzamenti a memoria della forma:

(base, indice, scala)

dove $base$ è l'indirizzo del primo byte dell'array, $indice$ è l'indice della cella dell'array che si vuole accedere, e $scala = \text{sizeof}(\text{elemento})$ è il numero di byte di ciascun elemento dell'array. Si noti che l'indirizzo effettivo $base + indice * scala$ calcolato dall'operando (base, indice, scala) realizza l'**aritmetica dei puntatori**, scalando l'indice in base alla dimensione degli elementi dell'array.

Se l'indice i dell'elemento che si vuole accedere è noto a tempo di compilazione, è possibile usare la forma:

`disp(base)`

dove `base` è l'indirizzo del primo byte dell'array e `disp=i*sizeof(elemento)` è lo spiazzamento in byte rispetto alla base dell'array per arrivare all'`i`-esimo elemento dell'array.

Esempio 1.

Si consideri la scrittura della cella `c`-esima dell'array `a` di `int`, assumendo che la variabile `a` sia tenuta in `eax` e la variabile `c` in `ecx`:

C da tradurre	C equivalente	Traduzione IA32
<code>a[c]=10;</code>	<code>*(a+c)=10;</code>	<code>movl \$10, (%eax,%ecx,4)</code>

Si noti che la scala è 4 poiché l'array è di `int` e `sizeof(int)==4`.

Esempio 2.

La seguente funzione C calcola la somma degli elementi di un array di due `int` passato come parametro:

C da tradurre	C equivalente	Traduzione IA32 (no <code>ebp</code>)
<pre>int sum(int c[2]) { return c[0]+c[1]; }</pre>	<pre>int sum(int c[2]){ int a = c[0]; a += c[1]; return a; }</pre>	<pre>sum: movl 4(%esp), %ecx movl (%ecx), %eax addl 4(%ecx), %eax ret</pre>

Si noti che in questo caso gli indici 0 e 1 nell'array `c` sono noti a tempo di compilazione (costanti nel codice) ed è quindi possibile calcolare gli spiazzamenti delle rispettive celle che si vogliono accedere (0 e 4).

Esempio 3.

Generalizziamo la funzione vista sopra per sommare gli elementi di un array `v` di dimensione arbitraria `n`. Assumendo di tenere l'indirizzo `v` dell'array in `ecx`, la dimensione `n` dell'array in `edx`, la somma `s` degli elementi di `v` in `eax`, e l'indice `i` per scorrere l'array in `ebx`, possiamo scrivere:

C da tradurre	C equivalente	Traduzione IA32 (no ebp)
<pre>int sum(int* v, int n){ int i, s=0; for (i=0; i<n; i++) s += v[i]; return s; }</pre>	<pre>int sum(int* v, int n){ int s = 0; int i = 0; L: if (i>=n) goto E; s += v[i]; i++; goto L; E: return s; }</pre>	<pre>sum: pushl %ebx # prologo movl 8(%esp),%ecx # ecx=v movl 12(%esp),%edx # edx=n movl \$0, %eax # eax=s movl \$0, %ebx # ebx=i L: cmpl %edx, %ebx jge E addl (%ecx,%ebx,4),%eax incl %ebx jmp L E: popl %ebx # epilogo ret</pre>

4 Come vengono eseguiti i programmi?

4.1 Processi

Quando un programma viene eseguito, esso dà luogo a un processo. Un **processo** è semplicemente un **programma in esecuzione**.

Uno **stesso programma** può essere istanziato in **più processi** che possono coesistere nel sistema. Ogni processo è identificato univocamente dal un **identificatore di processo** chiamato **PID** (Process ID). Il PID è un **numero progressivo** che viene incrementato di uno ogni volta che viene creato un nuovo processo.

Un processo è caratterizzato da principalmente da:

- Un'**immagine di memoria** che contiene il codice del programma e i dati da esso manipolati (variabili, blocchi allocati dinamicamente, ecc.)
- Lo **stato della CPU** (registri interni, ecc.)
- Un insieme di **risorse in uso** (file aperti, ecc.)
- Un insieme di **metadati** che tengono traccia vari aspetti legati al processo stesso e all'esecuzione del programma (identificatore del processo, utente proprietario del processo, per quanto tempo il processo è stato in esecuzione, ecc.)

Un processo può essere attivato in vari modi:

- su **richiesta esplicita dell'utente** che richiede l'esecuzione di un programma: questo può avvenire sotto forma di comandi impartiti da **riga di comando** (si veda l'[Appendice C](#)), oppure via **interfaccia grafica** facendo clic sull'icona associata a un programma eseguibile.
- su **richiesta di altri processi**
- **in risposta ad eventi** come lo scadere di un timer usato per attività programmate nel tempo (es. aggiornamento periodico della posta elettronica).

Linux/macOS X

Per elencare tutti i processi correntemente attivi nel sistema è possibile usare il comando `ps -e`.

```
$ ps -e
```

4.2 Esecuzione dei programmi

In questo paragrafo discutiamo come hardware e sistema operativo gestiscono l'esecuzione dei programmi, fornendo supporto per i processi. Partiamo dall'unità elementare di esecuzione, l'istruzione macchina. Mostriamo poi come i microprocessori riescono a velocizzare l'esecuzione eseguendo simultaneamente più istruzioni. Infine, vediamo come il sistema operativo, con il supporto dell'hardware, permette l'esecuzione simultanea di più processi.

4.2.1 Esecuzione di una singola istruzione

Sebbene dal punto di vista del programmatore un'istruzione macchina sia vista come un'unità indivisibile di esecuzione, la sua esecuzione è in realtà suddivisa dall'hardware in diverse sotto-operazioni che coinvolgono le varie parti della CPU:

- unità di controllo
- registri
- unità aritmetico-logica
- interfaccia verso la memoria e l'I/O

Le CPU sono tipicamente organizzate in stadi che sono delegati a portare a termine l'esecuzione delle istruzioni. Riportiamo a titolo di esempio una sequenza di stadi, presa in prestito dai classici processori RISC¹⁴, che supporta un sottoinsieme del set IA32:

1. **Fetch**: l'istruzione corrente viene prelevata dalla memoria e viene calcolato l'indirizzo dell'istruzione che segue la corrente in memoria;
2. **Decode**: eventuali operandi immediati dell'istruzione vengono letti, così come eventuali registri di input dell'istruzione;
3. **Execute**: se richiesto dall'istruzione, l'unità aritmetico-logica esegue un'operazione;
4. **Memory**: se richiesto dall'istruzione, la memoria viene acceduta in lettura o scrittura;
5. **Write-Back**: se richiesto dall'istruzione, i registri di output vengono aggiornati.

Si noti che istruzioni diverse possono impegnare stadi diversi. Ad esempio, un'istruzione `movl $0, %eax` eseguirà `fetch`, `decode` (per prelevare l'operando immediato `$0`) e `write-back` (per scrivere il risultato in `%eax`), attraversando gli stadi `execute` e `memory` senza impegnarli. Un'istruzione `addl %eax, %ecx` eseguirà invece `fetch`, `decode` (per prelevare il contenuto di `%eax` e `%ecx`), `execute` (per effettuare la somma) e `write-back` (per scrivere il risultato in `%ecx`), attraversando lo stadio `memory` senza impegnarlo. Osserviamo che un'istruzione come `addl $10, (%eax)` non può essere eseguita con gli stadi descritti poiché la lettura della memoria

¹⁴ I processori **RISC** (Reduced Instruction Set Computer) come MIPS, PowerPC, SPARC e ARM (usato nei dispositivi mobili come gli smartphone) hanno un set di istruzioni ridotto pensato per avere una latenza ridotta. I processori **CISC** (Complex Instruction Set Computer) come la famiglia x86 hanno istruzioni complesse che possono avere una latenza più alta, ma richiedono meno istruzioni dei RISC per svolgere un dato compito.

viene dopo l'esecuzione della somma. In architetture reali come i moderni processori Intel si può arrivare a 20 stadi e oltre.

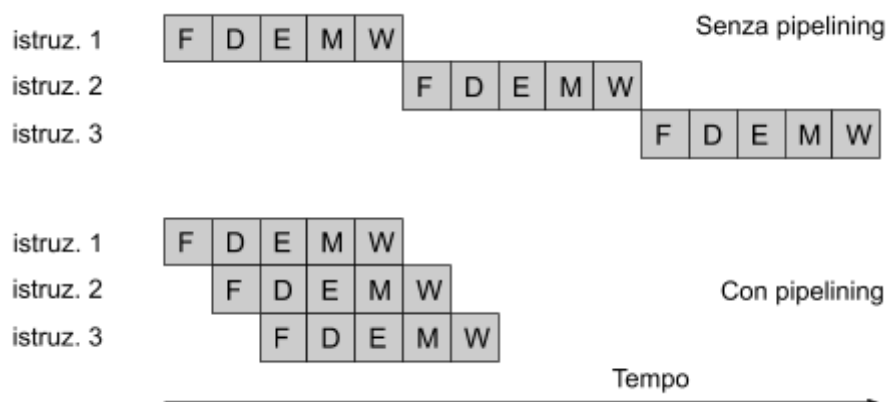
Ciascuno stadio richiede tipicamente qualche **centinaio di picosecondi** (10^{-12} sec). La sequenza di stadi viene ripetuta per ogni istruzione eseguita. Per scandire il tempo e consentire l'attivazione dei circuiti viene usato un orologio (**clock**) che crea un segnale elettrico come quello illustrato nella figura seguente:



In principio, con un design accurato, l'intera sequenza di stadi illustrata potrebbe essere completata in un unico ciclo di clock, con il fronte ascendente del segnale che fa da innesco allo stadio di fetch. La **frequenza di clock**, vale a dire il numero di oscillazioni al secondo (**cicli di clock**) del segnale, misura pertanto la velocità con cui vengono eseguite le istruzioni.

4.2.2 Esecuzione simultanea di più istruzioni: pipelining

L'idea del pipelining, concepita negli anni '60, nasce dall'osservazione che i diversi stadi impegnano porzioni diverse della circuiteria della CPU. In principio quindi, quando un'istruzione lascia uno stadio di fetch per entrare in quello di decode, si potrebbe immediatamente procedere a caricare una nuova istruzione, come in una catena di montaggio. Questo consente di ottenere un certo grado di parallelismo in cui più di un'istruzione è in esecuzione simultanea (**Instruction-Level Parallelism, IPL**). Nella figura seguente mettiamo a confronto l'esecuzione sequenziale vista nel paragrafo precedente con una con pipelining:



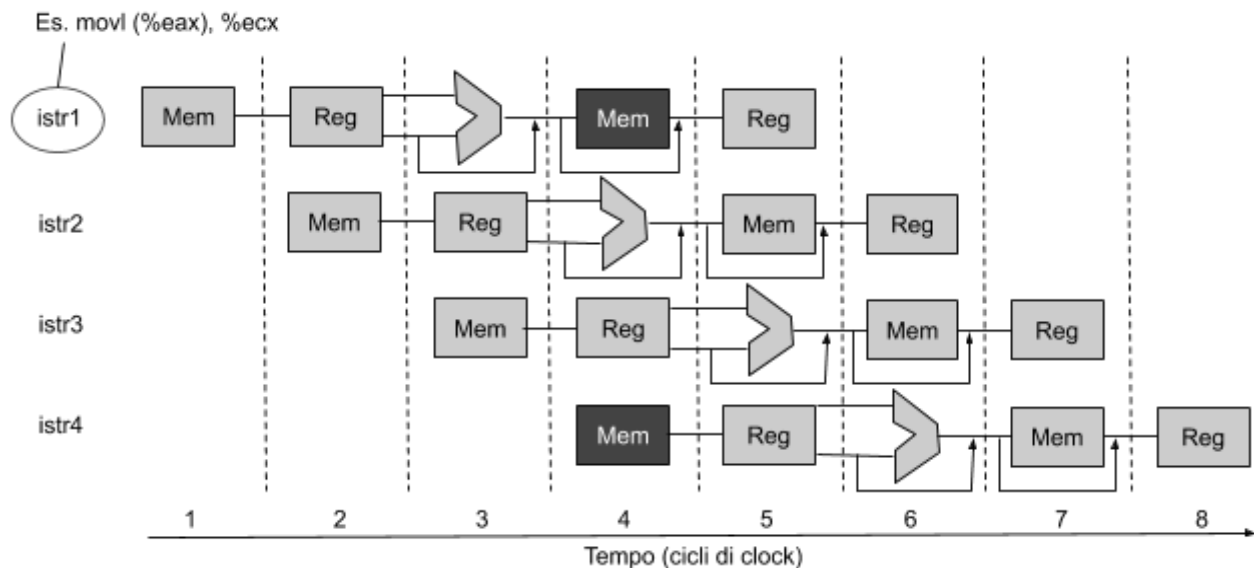
Si noti che il tempo per istruzione (**latenza**) non cambia (anzi, potrebbe leggermente crescere per via della maggiore complessità dell'hardware con pipelining). Si ha un impatto invece sul numero di istruzioni eseguite nell'unità di tempo (**throughput**), che in condizioni ideali può aumentare di un fattore pari al numero di stadi.

Diversamente dal caso sequenziale, la temporizzazione di una CPU con pipelining assume che ogni stadio richieda un ciclo di clock, con frequenze di clock più alte.

Purtroppo, vi sono varie ragioni (**hazards**) per cui non è sempre possibile avere una pipeline sempre a regime con le istruzioni in esecuzione durante il ciclo designato e ottenere quindi il throughput ideale:

1. **Hazard strutturali**: due o più istruzioni richiedono di usare simultaneamente lo stesso componente hardware come ad esempio la memoria;
2. **Hazard sui dati**: vi sono dipendenze fra i dati come ad esempio un'istruzione che ha come input l'output di quella precedente;
3. **Hazard sul controllo**: l'esecuzione di un'istruzione condizionale necessita una predizione (**branch prediction**) sul ramo che verrà preso per poter continuare a caricare istruzioni. Se la predizione è errata (**branch misprediction**) è necessario svuotare la pipeline e ripartire con la sequenza di istruzioni giusta.

Hazard strutturali. Nella figura seguente vediamo un esempio di hazard strutturale, evidenziando i componenti hardware in uso per ciascuno stadio:

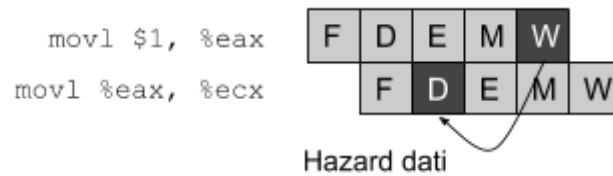


Si noti come al ciclo di clock 4 due istruzioni chiedono di accedere contemporaneamente alla memoria: istr1 per leggere un dato, istr4 per prelevare l'istruzione. Questo particolare tipo di hazard è in realtà mitigato dalla presenza di memorie tampone ([cache](#)) distinte per dati e istruzioni. Hazard strutturali possono coinvolgere anche altri componenti (registri, ALU) e possono essere in genere risolti duplicando l'hardware oppure mediante la tecnica dello stallo illustrata di seguito.

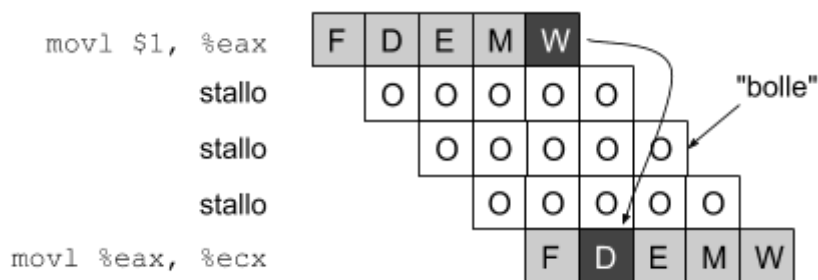
Hazard sui dati. Un esempio di hazard sui dati potrebbe essere il seguente:

```
movl $1, %eax
movl %eax, %ecx
```

Si noti che l'istruzione `movl %eax, %ecx` non può essere eseguita prima che `movl $1, %eax` abbia ultimato lo stadio di Write-Back:



Questo tipo di hazard viene normalmente risolto dall'hardware, con una penalità sul tempo di esecuzione, mettendo in **stallo** la pipeline tramite l'inserimento di "**bolle**" che prendono il posto delle istruzioni senza però impegnare nessuno stadio:



Un altro modo di mitigare gli hazard sui dati, effettuato non dall'hardware ma dal compilatore o dal programmatore che scrive direttamente in assembly, si ha modificando l'ordine delle istruzioni tramite una tecnica di ottimizzazione chiamata **instruction scheduling**. Consideriamo il seguente esempio:

```

movl $1, %eax
movl %eax, %ecx
movl $2, %esi
movl $3, %edi
movl $4, %edx

```

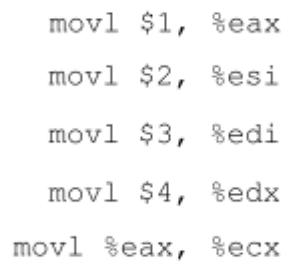
Come visto, le prime due istruzioni provocano uno stallo di tre cicli di clock. Riordiniamo ora le istruzioni come segue:

```

movl $1, %eax
movl $2, %esi
movl $3, %edi
movl $4, %edx
movl %eax, %ecx

```

L'effetto sulla pipeline è l'eliminazione completa degli stalli, come illustrato di seguito:



```
testl %eax, %eax
je L
movl %ecx, %eax
L: ...
```

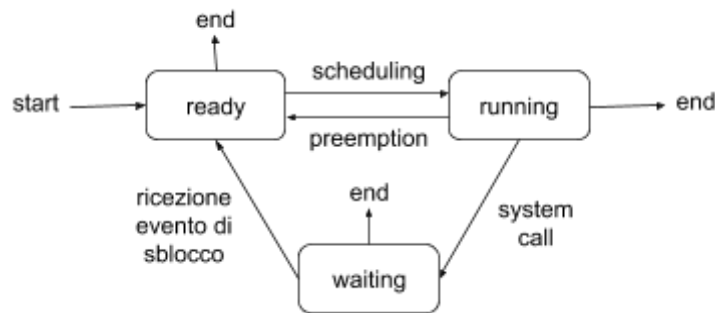
```
testl %eax, %eax
cmovne %ecx, %eax ← istruzione branchless
...
```

I sistemi operativi moderni permettono di avere più di un processo in vita allo stesso tempo. Questa caratteristica viene chiamata **multiprogrammazione**. Come vedremo nel prossimo paragrafo, ogni processo si trova in ogni istante in un determinato stato.

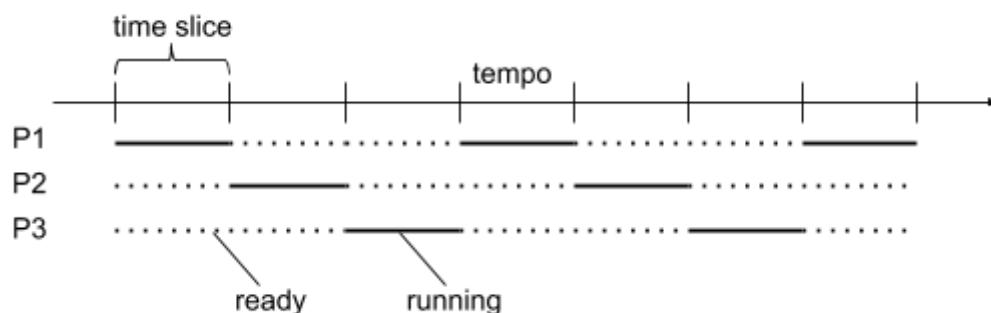
Gli **stati principali** di un processo in un sistema multiprogrammato sono in genere tre:

- 55

Il seguente diagramma mostra gli stati e le transizioni di stato effettuate dal sistema operativo:



Un processo entra tipicamente in uno stato di attesa (**waiting** o **sleeping**) volontariamente eseguendo una **system call** che può richiedere potenzialmente un'attesa prolungata come ad esempio un'operazione di accesso alla rete o al disco. Quando questa operazione si conclude, il processo si sblocca ed entra nello stato **ready**. L'operazione di **schedulazione** seleziona un processo **ready** per l'esecuzione. La **preemption** è una caratteristica dei sistemi operativi moderni che consente di sottrarre un core della CPU a un processo **running** per darlo a un altro processo. La **preemption** consente di contrastare il fenomeno della **starvation**, dove un processo rimane in attesa perenne senza progredire nella computazione perché altri processi tengono per sé tutti i core della CPU. La **preemption** viene applicata nei sistemi operativi **time-sharing** come Linux, MacOS X e Windows, che consentono di eseguire più processi del numero di core disponibile, dando l'illusione all'utente che i processi vengano eseguiti simultaneamente. Questo avviene assegnando a ogni processo **running** una **time slice**, ovvero un intervallo di tempo in cui hanno un core dedicato per l'esecuzione, alla fine della quale avviene **preemption** e viene schedulato per l'esecuzione un altro processo **ready**. Valori tipici per una **time slice** sono dell'ordine dei **millisecondi**. Per consentire a tutti i processi **ready** di progredire uniformemente nell'esecuzione, un popolare algoritmo di schedulazione è il **round-robin**, che assegna un core a rotazione a ciascun processo **ready** per la durata di una **time slice**. In pratica è spesso possibile assegnare una **priorità** a un processo in modo che venga schedulato per l'esecuzione più frequentemente degli altri. Il funzionamento dell'algoritmo di schedulazione **round-robin** è illustrato nel seguente diagramma a tre processi con la stessa priorità:



Come si vede, l'algoritmo **round-robin** non genera **starvation** nei processi, che hanno tutti l'opportunità di progredire nella computazione. Il sistema operativo tiene tipicamente i **PCB** dei processi organizzati in **liste separate per ciascuno stato**. Ad esempio, vi sarà una lista di processi **ready**, una lista di processi **waiting**, ecc. In un sistema operativo reale, il numero di stati

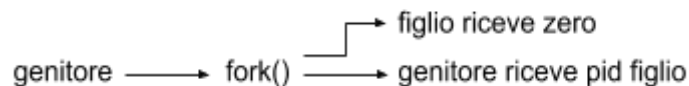
può essere maggiore per distinguere ad esempio situazioni in cui un processo sia ready in modalità utente o in modalità kernel (se stava eseguendo una system call). Vedremo altri stati nel seguito della dispensa.

4.2.3.2 Creazione di processi: fork

I sistemi operativi che aderiscono alla famiglia di standard POSIX anche parzialmente come Linux offrono una batteria di system call per la gestione di processi. La principale system call è la `fork`:

<pre>#include <unistd.h> pid_t fork();</pre>
La chiamata genera un nuovo processo figlio ottenuto come clone di quello genitore che esegue la <code>fork</code> . In uscita dalla chiamata a <code>fork</code> , sia il processo figlio che quello genitore proseguono l'esecuzione in modo indipendente dall'istruzione che segue la <code>fork</code> . E' possibile distinguere il genitore dal figlio tramite il valore restituito dalla <code>fork</code> .
Parametri: <ul style="list-style-type: none">nessuno
Risultato: <ul style="list-style-type: none">pid del processo figlio, per il processo genitore0 per il processo figlio-1 in caso di errore, descritto dalla variabile <code>errno</code>

La semantica della system call `fork` è diversa da quella di una normale chiamata a funzione, dove lo stesso processo è in esecuzione prima e dopo la chiamata.



Vediamo un semplice esempio di uso della `fork`:

<pre>#include <stdio.h> #include <stdlib.h> #include <unistd.h> int main() { pid_t pid = fork(); if (pid == -1) { perror("fork"); exit(EXIT_FAILURE); } if (pid == 0) { printf("sono nel processo figlio\n"); } else { printf("sono nel processo genitore\n"); } }</pre>

```

    }
    return EXIT_SUCCESS;
}

```

Eseguendo il programma si ottengono entrambe le stampe "sono nel processo figlio" e "sono nel processo genitore". Questo non viola la semantica di mutua esclusione del costrutto `if ... else` poiché dopo la `fork` i processi in esecuzione sono due e ciascuno esegue un ramo diverso dell'`if ... else`.

Il linguaggio C permette al programmatore di impostare delle operazioni che devono essere automaticamente effettuate alla terminazione del programma, ad esempio per rilasciare in modo ordinato le risorse in caso di errore. Si pensi a un programma che manipola file che devono essere lasciati in uno stato consistente in caso di terminazione prematura. Questo può essere fatto mediante la chiamata `atexit`, che installa un gestore che viene eseguito quando si esce dal `main` con `return` o viene chiamata `exit`. Tuttavia, il gestore non viene invocato quando il programma termina con `_exit`. Poiché il processo figlio creato da `fork` è un clone del genitore, eseguirà anch'esso eventuali gestori installati con `atexit` con il rischio di generare inconsistenze. Per evitare che questo accada, è opportuno fare in modo che il processo genitore termini con `exit` (o `return` dal `main`) e il figlio con `_exit`.

4.2.3.3 Recupero del pid dei processi: `getpid`, `getppid`

Un processo può conoscere il proprio pid e quello del proprio genitore mediante le system call `getpid` e `getppid`, rispettivamente:

```

#include <sys/types.h>
#include <unistd.h>
pid_t getpid();
pid_t getppid();

```

Parametri:

- nessuno

Risultato:

- pid del processo per `getpid`
- pid del processo genitore per `getppid`
- le chiamate non falliscono mai e quindi non restituiscono mai un codice di errore

La relazione padre-figlio induce una gerarchia sui processi, che sono organizzati nei sistemi UNIX/Linux in un **albero dei processi** che ha come radice il processo `init`, il **primo processo** a partire all'avvio del sistema operativo.

4.2.3.4 Attesa della terminazione di un figlio: `wait`

Se un processo **genitore termina prima di un suo figlio**, il processo figlio assume come nuovo genitore un altro processo di sistema antenato di quello terminato, che può essere `init` stesso o

un suo discendente. Un problema che sorge ove un genitore termini prima di un suo figlio è che il genitore non è in grado di conoscere lo stato di terminazione del figlio, specificato con `return` nel `main` o con `exit`. Lo stato di terminazione potrebbe infatti segnalare una terminazione anomala del processo figlio.

Per evitare questo problema si ha la system call `wait`, che permette a un processo genitore di attendere la terminazione di un figlio e prelevarne lo stato di terminazione:

<pre>#include <sys/types.h> #include <sys/wait.h> pid_t wait(int* status_ptr);</pre>
Attende la terminazione di un processo figlio
Parametri: <ul style="list-style-type: none">• puntatore a un buffer in cui viene scritto lo stato di terminazione del processo figlio, oppure <code>NULL</code> se questa informazione non è di interesse.
Risultato: <ul style="list-style-type: none">• pid del processo figlio terminato• -1 in caso di errore, descritto dalla variabile <code>errno</code>

Se un processo figlio termina prima che il genitore abbia avuto l'opportunità di effettuare una `wait` per prelevarne lo stato di terminazione, il figlio diventa uno **zombie**. Un processo zombie rilascia tutte le risorse che erano in uso (memoria, CPU, file, ecc.) e conserva solo lo stato di terminazione in modo che possa essere recuperato dal genitore con `wait`.

Si noti che invocando `wait(&status)`, `status` non conterrà in generale esattamente il codice di terminazione restituito dal `return` del `main`, `exit`, o `_exit`. A questo scopo, in ambiente Linux esistono delle macro che consentono di recuperare in modo affidabile il codice effettivo di terminazione, che è un valore a 8 bit:

1. `WIFEXITED(status)`: restituisce 1 se il processo è terminato con `return` dal `main`, con `exit` o con `_exit`, e 0 altrimenti. Esistono altre cause di terminazione come la ricezione di un segnale, come vedremo più avanti.
2. `WEXITSTATUS(status)`: estrae gli 8 bit meno significativi da `status`, che contengono il codice di terminazione del figlio.

Esempio. Vediamo ora un esempio di uso di `wait` e delle macro `WIFEXITED` e `WEXITSTATUS` per estrarre lo stato di terminazione di un processo figlio:

<pre>#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <sys/types.h> #include <sys/wait.h></pre>
--

```

int main() {
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        printf("Processo figlio restituisce 25...");
        _exit(25);
    }
    int status;
    pid = wait(&status);
    if (pid == -1) {
        perror("wait");
        exit(EXIT_FAILURE);
    }
    if (WIFEXITED(status))
        printf("Il figlio ha restituito %d\n", WEXITSTATUS(status));
    return EXIT_SUCCESS;
}

```

4.2.3.5 Caricamento di programmi: exec e invocazione del loader

Poiché `fork` crea un clone del processo genitore, da solo non consente di lanciare programmi eseguibili. A questo scopo è possibile utilizzare la famiglia di system call `exec`. Queste system call prendono tutte come primo parametro il pathname del file eseguibile da caricare e come altri parametri gli argomenti da passare all'eseguibile. Le system call `exec` invocano uno dei moduli più importanti del sistema operativo, il **loader**. Questo modulo trasforma il processo che invoca la `exec` eliminando tutte le sue sezioni (text, data, heap, stack, ecc.) e rimpiazzandole con quelle del programma eseguibile caricato. In una `exec` si entra da un processo con una certa immagine di memoria logica e si esce con un'immagine di memoria logica completamente diversa, riprendendo l'esecuzione dalla prima istruzione della funzione `_start` del programma eseguibile lanciato. Per questo motivo, se `exec` ritorna al programma chiamante lo fa sempre con un codice di errore che segnala l'impossibilità di caricare il nuovo programma. Il pid del processo che invoca una `exec` rimane lo stesso durante la trasformazione. Della famiglia `exec`, discutiamo solo la `execvp`, illustrata di seguito.

```

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

```

Rimpiazza l'immagine del processo corrente con quello di un nuovo programma eseguibile

Parametri:

- `file`: nome del file eseguibile da caricare. Il nome del programma viene cercato in tutti i percorsi di ricerca specificati dalla variabile di ambiente `PATH`.
- `argv`: array di stringhe terminato con `NULL` che contiene gli argomenti da passare al `main` del programma, dove `argv[0]` deve essere il nome del programma eseguibile.

Risultato:

- -1 in caso di errore, descritto dalla variabile `errno`

Esempio. Mostriamo ora un esempio di programma che lancia il comando `ls -l` per elencare i file nella directory corrente:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        char* argv[] = { "ls", "-l", NULL };
        execvp("ls", argv);
        perror("execvp");
        _exit(1);
    }
    pid = wait(NULL);
    if (pid == -1) {
        perror("wait");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

4.3 Flusso del controllo eccezionale

Un programma normalmente segue un flusso del controllo determinato dalle istruzioni di salto o semplicemente dal susseguirsi delle istruzioni in memoria. Si parla invece di **flusso del controllo eccezionale** ogni volta che il program counter (registro EIP in IA32) di un core della CPU assume un valore che sposta l'esecuzione in una zona di codice non conforme con il normale flusso del controllo. Il flusso del controllo eccezionale viene spesso utilizzato per gestire situazioni anomale, ma non solo. Le **eccezioni**, nei linguaggi che le supportano (es. mediante i costrutti `throw`, `try`, `catch`), sono un esempio di flusso di controllo eccezionale che consente di spostare la computazione in opportune sezioni di codice di gestione di eventi, generalmente imprevisti (`catch`).

In questo paragrafo vedremo altri due classici modi di generare un flusso di controllo eccezionale per gestire in maniera pulita un ampio spettro di eventi che sorgono in un sistema di calcolo: le

interruzioni (interrupt), gestite con il supporto dell'hardware, e i **segnali**, forniti come servizio di comunicazione tra processi dal sistema operativo.

4.3.1 Interruzioni

Un'**interruzione (interrupt)** è una notifica inviata alla CPU via hardware o eseguendo istruzioni software per segnalare un evento che richiede immediata attenzione da parte del sistema di calcolo. Un interrupt interrompe il programma correntemente in esecuzione su uno dei suoi core per eseguire un codice di gestione dell'interruzione impostato dal sistema operativo. Le interruzioni sono utilizzate tipicamente per segnalare eventi come clic del mouse, pressione di un tasto sulla tastiera, terminazione di operazioni di scrittura o lettura da disco, eventi anomali come una divisione per zero o un accesso errato a memoria, e molto altro. A seconda del tipo di interruzione, il programma interrotto può riprendere la propria esecuzione da dove era stata interrotta o meno.

I diversi tipi di interruzioni sono identificati da numeri che vengono usati per indicizzare un array di puntatori (**interrupt vector table**¹⁵) ai rispettivi codici di gestione. L'interrupt vector table non è accessibile ai programmi utente e viene impostato dal sistema operativo. Con il termine interrupt vector si intende l'indirizzo del codice di gestione di un'interrupt.

Le interruzioni si dicono **recuperabili** se è possibile continuare l'esecuzione del programma interrotto, e non recuperabili altrimenti. Un'interruzione è **intenzionale** se è determinata volutamente dal programma, e non intenzionale altrimenti.

Possiamo classificare le interruzioni in due grandi categorie:

- **Interruzioni asincrone**: sono generate dall'hardware inviando un segnale elettrico (IRQ, Interrupt Request) alla CPU mediante uno dei piedini metallici che la connettono alla scheda madre. Ricadono in questa categoria le interruzioni generate dai dispositivi esterni. Esempi: pressione di CTRL+C o CTRL+ALT+DEL (soft reset) sulla tastiera, arrivo di un pacchetto dati sulla scheda di rete, completamento di un'operazione su disco, pressione del bottone di reset del PC (hard reset), clic del mouse, e molto altro.
- **Interruzioni sincrone**, che si dividono a loro volta in tre sottocategorie¹⁶:
 - a. **trap** (interruzioni software): sono generate intenzionalmente da un programma eseguendo un'apposita istruzione prevista dal set della CPU. In IA32, una trap si realizza ad esempio mediante l'istruzione `INT k`, dove `k` è il numero dell'interruzione. In particolare, l'istruzione `INT $0x80` viene utilizzata per passare il controllo al sistema operativo richiedendo l'esecuzione di una funzione di servizio (system call, o chiamata a sistema). Nel caso di trap, l'esecuzione riprende dall'istruzione successiva alla trap.
 - b. **fault**: sono interruzioni non intenzionali generate dalla CPU, sono generalmente recuperabili e dopo la loro gestione da parte del kernel il programma riprende ripetendo l'esecuzione dell'istruzione in cui si è generato il fault. Ad esempio il

¹⁵ Interrupt Descriptor Table (IDT) nei sistemi x86.

¹⁶ La terminologia relativa alle interruzioni può variare a seconda dell'architettura di macchina considerata. ad esempio, in alcuni casi il termine trap viene usato per indicare alcuni tipi di fault.

tentativo di lettura o scrittura a una zona di memoria a cui il programma non è consentito accedere genera un fault (page fault).

- c. **abort**: sono interruzioni non intenzionali generate dalla CPU e sono sempre non recuperabili. In alcuni casi, bloccano la CPU e richiedono il riavvio della macchina. Ad esempio, un errore nella diagnostica interna della CPU che evidenzia un malfuoramento può generare un abort.

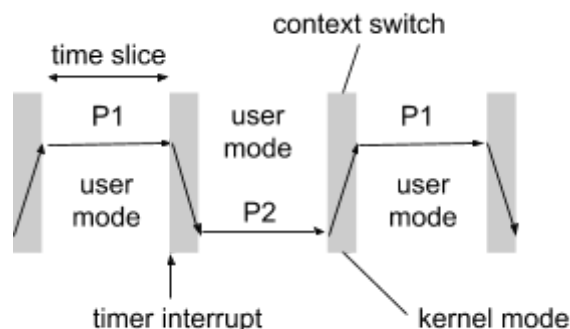
Vediamo ora alcuni esempi di operazioni rilevanti in un sistema di calcolo gestite mediante interruzioni.

4.3.1.1 Preemption e context switch tra processi

I sistemi multitasking con time sharing si basano su interrupt generati da un timer per realizzare la preemption: allo scadere di una time slice, tipicamente della durata dell'ordine di grandezza dei millisecondi, il timer interrompe l'esecuzione di un processo correntemente in esecuzione passando il controllo al kernel con il tramite dell'interrupt vector table. Il kernel seleziona un processo ready in base a un determinato algoritmo di scheduling (es. round-robin) e avviene una commutazione di contesto (**context switch**) che consiste dei seguenti passi:

1. lo stato del processo interrotto (contenuto corrente dei registri, ecc.) viene salvato in un'opportuna sezione del PCB del processo in modo da consentire in futuro il ripristino dell'esecuzione dall'esatto punto dove il processo è stato interrotto;
2. il PCB del processo interrotto viene inserito dal kernel nella coda dei PCB dei processi ready;
3. lo stato del processo ready da portare in esecuzione viene impostato prendendolo dal PCB. Lo stato potrebbe essere quello iniziale per i nuovi processi, oppure uno stato precedentemente salvato durante un context switch passato.

La seguente immagine illustra uno scenario in cui due processi P1 e P2 si alternano in time sharing nell'uso di un core della CPU. I context switch avvengono allo scadere della time slice e sono innescati dall'interrupt di un timer. Si noti che l'esecuzione procede in modalità utente durante la time slice (a meno di chiamate a sistema o altre interruzioni) e in modalità kernel durante i context switch:



Le operazioni svolte durante un context switch sono **puro overhead** per il sistema, poiché non contribuiscono a svolgere compiti utili alla risoluzione dei problemi di calcolo affrontati dai processi. Tuttavia, consentendo il time sharing, migliora l'esperienza dell'utente nell'interazione

con il sistema dando l'illusione di poter avere più processi simultaneamente in esecuzione del numero di core disponibili nella CPU. Per minimizzare il tempo sprecato nei context switch i progettisti dei sistemi operativi impostano la durata della time slice in modo da non essere troppo piccola, il che renderebbe troppo frequenti i context switch (che hanno un tempo di gestione fisso), ma neanche troppo grande per rendere percepibile all'utente che i processi vengono intervallati e non sono realmente eseguiti in parallelo.

4.3.1.2 System call

Un'altra classica applicazione delle interruzioni sono le chiamate a sistema. A questo scopo, si fa spesso uso di interrupt sincroni basati su trap. Il vantaggio è la possibilità di cambiare modalità di esecuzione passando da utente a supervisore, consentendo di eseguire porzioni di codice del kernel in modo sicuro e controllato come se fossero chiamate a funzione, ma con tutti i privilegi richiesti per portare a termine l'operazione.

Nei sistemi Linux IA32¹⁷, una chiamata a sistema si realizza mediante l'istruzione `INT $128` (o `INT $0x80`), dopo aver predisposto un certo numero di registri:

<code>eax</code>	numero identificativo della system call ¹⁸
<code>ebx</code>	1° argomento
<code>ecx</code>	2° argomento
<code>edx</code>	3° argomento
<code>esi</code>	4° argomento
<code>edi</code>	5° argomento
<code>ebp</code>	6° argomento

L'eventuale valore restituito dalla system call è memorizzato in `eax`. Per fornire al programmatore un'interfaccia C che consenta di invocare una system call come se fosse una normale funzione, si fa uso di funzioni wrapper scritte in assembly che hanno l'unico scopo di impostare i parametri della system call e invocarla.

Esempio. Vediamo come scrivere un wrapper per la system call `int exit(int)`:

```
.globl exit
exit:                                # funzione int exit(int)
    pushl %ebx
```

¹⁷ Nei sistemi a 64 bit si usa un'istruzione assembly diversa e più efficiente (`syscall`) che non richiede l'accesso all'interrupt vector table.

¹⁸ In Linux, i numeri delle system call sono generalmente reperibili nella header: `/usr/src/linux-headers-xxx-generic/arch/x86/include/generated/uapi/asm/unistd_32.h`, dove xxx è un numero di versione, ad esempio 3.13.0-95.

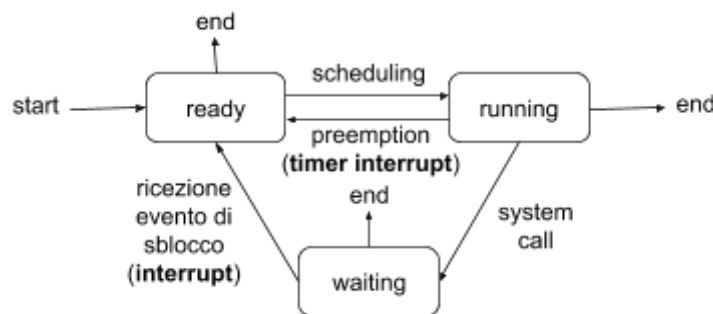

```

movl $1, %eax      # imposta il numero della system call
movl 8(%esp), %ebx  # passa parametro alla exit
int $0x80          # invoca la exit
popl %ebx
ret

```

4.3.1.3 Passaggio di stato di un processo da waiting a ready: interrupt vs. polling

Quando un processo P entra in attesa di un evento, liberando la CPU in modo che sia disponibile per l'esecuzione di altri processi ready, si pone il problema di come intercettare quell'evento in modo da rimettere il processo P in stato ready. La tecnica più efficace è basata sulle **interruzioni** asincrone. Questo avviene ad esempio nell'interfacciamento con dispositivi esterni come i dischi: se il processo P entra in stato waiting poiché ha chiesto mediante una system call di leggere o scrivere dei dati, un'interruzione generata dal dispositivo stesso segnerà il completamento dell'operazione e consentirà al kernel di rimettere il PCB di P nella lista dei processi ready. Un altro esempio è l'attesa realizzata mediante la system call `sleep`, che ferma il processo per un determinato intervallo di tempo allo scadere del quale si ha un'interruzione generata da un timer che risveglia il processo. La seguente figura rivisita il diagramma degli stati dei processi evidenziando il ruolo delle interruzioni nelle transizioni di stato.



Un'alternativa all'uso di interrupt è il meccanismo del **polling**, dove il processo non viene notificato dall'esterno, ma piuttosto verifica attivamente che l'evento atteso si sia verificato, ad esempio controllando un apposito bit impostato dal dispositivo. Una forma estrema di polling è il **busy waiting**, dove il processo rimane in un ciclo di esecuzione che fa polling dell'evento ad ogni iterazione. Se il tempo di attesa dell'evento è significativo, questo meccanismo è estremamente dispendioso in termini di risorse consumando prezioso tempo di CPU e non è generalmente adottato dai sistemi operativi moderni per innescare la transizione tra stato waiting e ready.

4.3.2 Segnali

Un segnale è una **notifica** inviata, generalmente in modo asincrono, a un processo. La differenza principale tra un segnale e un'interruzione è che un segnale è completamente gestito via software e ha come destinatario un processo e non la CPU. Un segnale può essere inviato dal kernel a un processo, da un processo a un altro processo, o da un processo a se stesso. Un segnale potrebbe essere ad esempio una richiesta di terminazione di un processo da parte di un altro processo, la notifica di un accesso a memoria errato, o la richiesta diretta dell'utente di terminare il

processo figlio del terminale correntemente attivo (anche detto in **foreground**) tramite la pressione dei tasti CTRL+C sul terminale, ecc. Quest'ultimo esempio mostra come un segnale possa essere prodotto in risposta a un'interruzione.

4.3.2.1 Definizione segnali

Un segnale è definito:

1. da un **numero** che lo identifica univocamente nel sistema usato e
2. dal **pid** del processo a cui deve essere notificato¹⁹.

Nonostante le molte similarità, si faccia attenzione a non confondere i segnali con le eccezioni, che sono notifiche dall'hardware verso il sistema operativo e hanno una gestione differente. Si noti tra l'altro che la numerazione degli interrupt e dei segnali sono del tutto diversi.

Salvo intervento del programmatore per ottenere un risultato diverso, ogni segnale, una volta notificato al processo a cui è rivolto, risulta in uno dei seguenti **comportamenti**²⁰ predefiniti:

Term	Il processo ricevente termina
Core	Il processo ricevente termina e genera un file chiamato <code>core dump</code> che consiste in un'istantanea dello stato della memoria del processo e dei registri per scopi di debugging al momento in cui il segnale viene ricevuto. L'analisi del core dump può essere utile per comprendere il motivo di un'eventuale situazione anomala che ha innescato l'invio del segnale.
Ign	Il segnale viene ignorato e non ha alcun effetto sul processo ricevente

La seguente tabella, non esaustiva, elenca i principali segnali trattati nella dispensa:

Catego ria	Segnale	N ²¹	Descrizione	Ignor abile	Gesti bile	Default
Termin azione	SIGTERM	15	Terminazione "morbida" in cui il processo ha la facoltà di liberare le proprie risorse. Inviato dal comando <code>kill</code> di default (si veda sotto).	Sì	Sì	Term
	SIGINT	2	Come SIGTERM, ma attivato anche tramite la combinazione di tasti CTRL+C	Sì	Sì	Term
	SIGQUIT	3	Come SIGTERM, ma attivato anche tramite la combinazione di tasti CTRL+\	Sì	Sì	Core
	SIGKILL	9	Terminazione forzata a cui il processo non può opporsi e liberare le risorse acquisite.	No	No	Term
	SIGCHLD	17	Evento terminazione di un figlio del processo	Sì	Sì	Ign

¹⁹ E' possibile inviare segnali a gruppi di processi, ma la trattazione va oltre gli scopi della dispensa.

²⁰ <http://man7.org/linux/man-pages/man7/signal.7.html>

²¹ Numerazione vigente su sistemi x86, ARM e altri.

			ricevente			
Mem.	SIGSEGV	11	Accesso errato a memoria da parte del processo stesso	Sì	Sì	Term
Expr.	SIGFPE	8	Errore aritmetico (es. divisione per zero)	Sì	Sì	Core
Timing	SIGALRM	14	Segnale che indica un evento temporale	Sì	Sì	Term
Msg.	SIGUSR1	10	Segnale liberamente utilizzabile dall'utente come messaggio	Sì	Sì	Term
	SIGUSR2	12	Segnale liberamente utilizzabile dall'utente come messaggio	Sì	Sì	Term

Come vedremo più avanti, "ignorabile" nella tabella significa che un processo può programmare quel segnale in modo che non abbia alcun effetto quando ricevuto. Per "gestibile" si intende la possibilità per il processo di installare un handler (gestore), ovvero una funzione che viene invocata ogni volta che viene ricevuto un segnale di quel tipo. Per mantenere la trattazione compatta, non trattiamo la possibilità per un processo di "bloccare" segnali di un certo tipo e la gestione di segnali multipli inviati simultaneamente a un processo, né altri argomenti più complessi.

4.3.1.2 Invio segnali

Fra le varie **system call** progettate per inviare programmaticamente un segnale a un processo [riportiamo](#):

<pre>#include <signal.h> int kill(pid_t pid, int sig);</pre>
Parametri: <ul style="list-style-type: none"> • <code>pid_t pid</code>²²: pid del processo a cui il segnale deve essere inviato. Il mittente è il processo che esegue la <code>kill</code>. • <code>int sig</code>: numero del segnale da inviare, fra quelli listati in <signal.h>. Per chiarezza, si raccomanda l'utilizzo della costante alfanumerica, ad esempio <code>SIGKILL</code> invece di 9. Nel caso speciale <code>sig=0</code>, <code>kill</code> non invia alcun segnale e verifica semplicemente la validità del <code>pid</code> destinatario.
Risultato: <ul style="list-style-type: none"> • 0 in caso di successo • -1 in caso di errore, descritto dalla variabile <code>errno</code>

Alla system call `kill` corrisponde l'omonimo comando da terminale `kill`:

<code>> kill -segnale pid</code>	<code>> kill -s segnale pid</code>
-------------------------------------	---------------------------------------

²² Omettiamo per brevità una serie di altre possibilità per `pid <= 0` come anche la gestione dei permessi che consentono a un processo di inviare un segnale a un altro.

Il comando invia un segnale al pid specificato. Il comando consente di inviare solo alcuni segnali come definito dal [manuale.opengroup](#). Le seguenti forme sono tutte equivalenti nel seguente esempio:

- `kill -s SIGTERM 8756`
- `kill -s 15 8756`
- `kill -SIGTERM 8756`
- `kill -15 8756`

Come anticipato, `kill pid` equivale a `kill -SIGTERM pid`.

Si noti che alcuni segnali, come `SIGSEGV`, vengono generati in risposta a un accesso errato a memoria e sono inviati dal kernel e non da altri processi.

4.3.1.4 Gestione segnali: `sigaction`

Salvo poche eccezioni (es. `SIGKILL`) è possibile ridefinire cosa accade al momento della ricezione di un segnale da parte di un processo. Ad esempio, si può fare in modo che un processo, prima di terminare per via di una `SIGTERM`, lasci i file in uso al processo in uno stato consistente. Storicamente, il linguaggio C fornisce a questo scopo la funzione [signal](#), di facile impiego per installare gestori per i segnali. La chiamata è tuttavia **deprecata** a favore della più articolata `sigaction`:

```
#include <signal.h>
```

```
int sigaction(int sig, const struct sigaction *act,  
              struct sigaction *oact);
```

Descrizione: la funzione consente di esaminare e/o specificare le azioni associate a uno specifico segnale

Parametri:

- `int sig`: numero del segnale da gestire, fra quelli listati in [<signal.h>](#)
- `struct sigaction act`: se `act` è diverso da `NULL`, viene impostata una nuova struttura che definisce il comportamento del segnale (si veda sotto), altrimenti il comportamento rimane invariato;
- `struct sigaction oact`: se il puntatore `oact` (si noti che `oact` fa riferimento a "old"-`act`) è diverso da `NULL`, l'attuale struttura che definisce il comportamento del segnale viene copiata in `oact`.

```
struct sigaction
```

Attributi:

- `void(*sa_handler)(int signum)`: gestore del segnale che può essere una delle seguenti varianti:
 - `SIG_IGN`: il segnale va ignorato

- `SIG_DFL`: viene ripristinato il gestore di default del segnale
- codice utente della forma `void handler(int signum)` che prende come parametro il numero del segnale ricevuto
- `int sa_flags`: impostazioni che influenzano il comportamento del segnale. Riportiamo un sottoinsieme di flag che, inseriti in OR tra loro, sono rilevanti per la dispensa:
 - `SA_RESTART`: se impostata, fa in modo che la system call o la funzione interrotta venga rieseguita da capo una volta gestito il segnale;
 - `SA_SIGINFO`: se **non** impostata, fa in modo che la gestione del segnale faccia capo all'attributo `sa_handler`. Vi è infatti una variante di gestore più potente `sa_sigaction` vista sopra che reputiamo al di là degli scopi di questa dispensa.²³ Le due varianti non possono però coesistere nella stessa `struct sigaction`.

Risultato:

- In caso di successo, restituisce 0 e non modifica le impostazioni per il segnale
- In caso di errore, restituisce -1 ed `errno` specifica i dettagli dell'errore

Esempio: il seguente programma mostra come ignorare il segnale `SIGINT` (CTRL+C) in modo che non provochi la terminazione del programma:

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

int main() {

    struct sigaction act = { 0 };           // preparazione struttura
    act.sa_handler = SIG_IGN;               // segnale ignorato
    int ret = sigaction(SIGINT, &act, NULL); // gestore installato
    if (ret == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    unsigned i = 0; // ciclo di esecuzione attiva
    while (i++ >= 0) {
        if (i % 100000000 == 0)
            printf("0xDEADBEEF...\n");
    }

    return EXIT_SUCCESS;                   // struct act ancora visibile
}
```

Nota importante: l'oggetto `struct sigaction` il cui puntatore viene passato alla chiamata `sigaction` deve rimanere in vita per tutta la durata della gestione del segnale. Il seguente esempio mostra un'applicazione errata dei segnali.

²³ Per maggiori informazioni sulla `sigaction` si rimanda al sito [opengroup](http://opengroup.org).

```

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void setup_signal() {
    struct sigaction act = { 0 };           // preparazione struttura
    act.sa_handler = SIG_IGN;               // segnale ignorato
    int ret = sigaction(SIGINT, &act, NULL); // gestore installato
    if (ret == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }                                       // errore: act va out of scope
}                                           // segnale non più catturato

int main() {
    setup_signal();                        // modularizzazione

    unsigned i = 0; // ciclo di esecuzione attiva
    while (i++ >= 0) {
        if (i % 100000000 == 0)
            printf("0xDEADBEEF...\n");
    }

    return EXIT_SUCCESS;
}

```

4.3.1.3 Attesa segnali: pause

Un altro modo di causare una risposta in un processo alla ricezione di un segnale è quello di bloccare il processo ricevente fino all'arrivo di un segnale. Questo può essere fatto in vari modi. Per semplicità discutiamo qui la system call `pause`:

```

#include <unistd.h>
int pause();

```

Parametri: nessuno. La chiamata sospende indefinitamente il processo fino all'eventuale arrivo di un segnale.

Risultato: poiché `pause` sospende il processo indefinitamente, non può mai terminare con successo. L'unico possibile valore di ritorno è pertanto -1, con `errno` impostato a `EINTR`.

La `pause` illustra un meccanismo generale la cui utilità pratica è tuttavia al di fuori del campo di trattazione della dispensa.

4.3.1.4 Timer

Uno strumento flessibile per la temporizzazione degli eventi in un programma consiste nel programmare segnali `SIGALRM` futuri, da inviarsi allo scadere di un certo quanto di tempo. Fra le innumerevoli possibilità, riportiamo le seguenti due opzioni:

```
#include <unistd.h>
unsigned alarm(unsigned seconds);24
```

Parametri: `seconds`. La chiamata programma un segnale dopo `seconds` secondi (tempo reale) e il programma continua normalmente. Scaduto il quanto di tempo, il processo invia a se stesso un segnale `SIGALRM`, che ha come default la terminazione del processo. Se una seconda chiamata ad `alarm` viene effettuata prima che la prima abbia inviato il segnale `SIGALRM`, la seconda chiamata prende il sopravvento sulla prima, che viene annullata. Se una seconda chiamata con `seconds=0` viene emessa, questa annulla la precedente.

Risultato:

- se vi è una sola chiamata `alarm` alla volta, restituisce zero;
- se vi è una precedente chiamata ancora non conclusa con la consegna del segnale, restituisce il tempo mancante per l'emissione di quel segnale;
- non vi sono situazioni possibili di errore.

```
#include <unistd.h>
useconds_t ualarm(useconds_t useconds, useconds_t interval);25
```

Parametri:

- `useconds`. La chiamata programma un primo segnale `SIGALRM` dopo `useconds` microsecondi (tempo reale) e il programma continua normalmente. Scaduto il quanto di tempo, il processo invia a se stesso un segnale `SIGALRM`, che ha come default la terminazione del processo.
- `interval`: se diverso da zero, programma una successiva sequenza di segnali `SIGALRM` emessa ogni `interval` microsecondi.

Risultato:

- il numero di microsecondi dalla precedent chiamata a `ualarm`, o zero altrimenti
- non vi sono situazioni possibili di errore

Si rimanda alla documentazione [opengroup](http://pubs.opengroup.org/onlinepubs/007904875/functions/alarm.html) per la discussione della granularità dei timer.

²⁴ <http://pubs.opengroup.org/onlinepubs/007904875/functions/alarm.html>

²⁵ <http://pubs.opengroup.org/onlinepubs/007904875/functions/ualarm.html>

5 Come vengono gestiti i dati in memoria centrale?

5.1 Come viene allocata la memoria?

La memoria è vista generalmente da un processo come uno **spazio di indirizzi contigui** contenente codice, variabili, blocchi allocati dinamicamente, stringhe letterali, ecc. Il problema di gestire uno spazio di indirizzi di memoria tenendo traccia di quali zone sono **libere** e quali invece sono **in uso** ai processi è chiamato **allocazione della memoria**, ed è uno dei problemi più vecchi e fondamentali dell'informatica.

5.1.2 Allocazione dinamica della memoria

Un **allocatore di memoria** è una **struttura di dati** per la gestione della memoria. Un allocatore supporta generalmente tre primitive che possono essere usate dai programmi per richiedere blocchi di memoria da utilizzare e rilasciarli quando non servono più:

- `p=alloca(n)`: restituisce l'indirizzo iniziale di un blocco di almeno n byte contigui che possono essere usati per memorizzare dati e programmi;
- `p=ridimensiona(p,n)`: ridimensiona il blocco p ²⁶ alla nuova dimensione n e restituisce il nuovo indirizzo del blocco (che potrebbe essere cambiato);
- `dealloca(p)`: rilascia il blocco p .

Esempio. Quando un programma ha bisogno di spazio di memoria di appoggio per memorizzare i suoi dati, chiama `malloc()` (`alloca`). Se necessario, il programma può chiedere di ridimensionare un blocco con `realloc()` (`ridimensiona`). Quando quello spazio non serve più, chiama `free()` (`dealloca`).

Un classico errore di programmazione consiste nel non deallocare un blocco quando non è più in uso. Questo errore provoca **memory leak**, cioè l'accumularsi di blocchi considerati come in uso dall'allocatore - e quindi non utilizzabili per altri scopi, ma non effettivamente utilizzati dai programmi. Un memory leak è un bug molto serio che può portare all'**esaurimento progressivo della memoria disponibile**.

5.1.2.1 Frammentazione interna ed esterna

La **frammentazione** della memoria è un fenomeno per cui esiste spazio libero, ma non è utilizzabile per soddisfare le richieste dei programmi. Si hanno due tipi principali di frammentazione della memoria:

- **Frammentazione interna:** quando vi è spazio inutilizzato all'interno di un blocco precedentemente allocato. Ad esempio, il padding inserito dal compilatore per garantire

²⁶ Per semplicità, ci riferiamo al "blocco p " come al blocco il cui indirizzo iniziale è p .

allineamento è una fonte di frammentazione interna. Come altro motivo di frammentazione interna, gli allocatori alle volte riservano più memoria di quella richiesta. Questo avviene ad esempio se si richiede di allocare un blocco di dimensione inferiore a una certa soglia minima (es. 16 byte), dato che l'allocatore restituisce blocchi di dimensione non inferiore alla soglia.

- **Frammentazione esterna:** si manifesta se una **richiesta di allocazione** non può essere soddisfatta con lo **spazio libero disponibile non perché non sia sufficiente, ma perché non è contiguo**. La frammentazione esterna può aversi solo nel caso in cui viene richiesta l'allocazione di **blocchi di dimensioni diverse**. Se infatti i blocchi allocati sono tutti della stessa dimensione, quando vengono rilasciati lasciano spazio libero contiguo sempre utile per essere riutilizzato per soddisfare richieste future di allocazione.

5.1.2.2 Qualità di un allocatore: tempo e spazio

La bontà di un allocatore di memoria si misura in vari modi. Due delle qualità principali (ma ce ne sono molte altre) sono:

1. **tempo:** capacità di supportare il più velocemente possibile le operazioni di allocazione, ridimensionamento e deallocazione;
2. **spazio:** capacità di riusare il più possibile lo spazio precedentemente deallocato per soddisfare richieste di blocchi.

In genere, è facile scrivere un allocatore veloce, ma è molto difficile usare lo spazio in modo efficace. Il **problema** principale di un allocatore è la **mancaanza di informazioni sulle allocazioni/deallocazioni future**. Se un allocatore potesse aspettare di aver accumulato sufficiente informazioni su quanti blocchi e di che dimensioni sono necessari a un programma, potrebbe usare lo spazio in modo molto più efficace. Questo non è però possibile, visto che un'operazione di allocazione deve restituire un blocco allocato **immediatamente** quando viene richiesto.

5.1.2.3 Allocazione in cascata della memoria

In un sistema di calcolo, un blocco di memoria reso disponibile da un allocatore può essere a sua volta suddiviso in blocchi più piccoli da un altro allocatore appositamente adibito alla sua gestione. Possono pertanto aversi diversi allocatori in cascata, ognuno dei quali gestisce un blocco restituito dall'altro.

In un sistema multi-programmato, in cui cioè più processi possono coesistere, la memoria fisica deve essere allocata ai processi in modo che ciascuno abbia spazio per la propria immagine (codice, dati, heap, stack). Un **primo tipo di allocatore**, parte del sistema operativo, alloca spazio di memorizzazione fisico ai processi.

A sua volta, i processi devono poter gestire i rispettivi spazi heap, fornendo ai programmi spazio per blocchi allocati dinamicamente durante l'esecuzione. Un **secondo tipo di allocatore**, di cui ne esiste un'istanza diversa nello spazio utente di ciascun processo, gestisce lo spazio heap

mediante le primitive `malloc` e `free`. I blocchi restituiti da `malloc` contengono generalmente oggetti di tipi primitivi o composti (`struct`, `array`).

Alcune applicazioni usando un **terzo tipo di allocatore** (allocatore custom) che partiziona alcuni dei blocchi restituiti da `malloc` in blocchi più piccoli per consentire un uso ottimizzato dello spazio e migliori prestazioni.

5.1.1 Memoria fisica e memoria virtuale

Nei sistemi di calcolo più semplici, come alcuni sistemi embedded (es. modem/router, macchine fotografiche digitali, hardware di controllo degli elettrodomestici, ecc.), i processi indirizzano direttamente la memoria fisica. Ogni puntatore di un programma contiene un indirizzo di memoria fisica e tutti i processi condividono lo stesso spazio di indirizzi. Questo semplice schema soffre di vari problemi:

- **Mancanza di protezione.** Poiché tutti i processi "vedono" tutta la memoria fisica, un processo potrebbe erroneamente o maliziosamente²⁷ accedere alla parte di memoria in uso a un altro processo, compromettendone il funzionamento.
- **Frammentazione esterna.** Poiché lo spazio di memoria fisica va suddiviso fra processi che vengono continuamente creati ed eliminati, e ciascun processo ha bisogno di uno spazio **contiguo** di memoria, potrebbe crearsi frammentazione esterna con ampio spazio disponibile, ma sminuzzato in tante piccole zone libere insufficienti a soddisfare le necessità di memoria dei processi.

Per risolvere questi problemi, nei sistemi più complessi viene utilizzata una tecnica più raffinata chiamata **memoria virtuale**, in cui ogni processo ha un suo **spazio virtuale di indirizzi** (anche chiamato **spazio logico di indirizzi**) distinto da quello fisico.

5.1.3 Allocazione nella memoria fisica: memoria virtuale

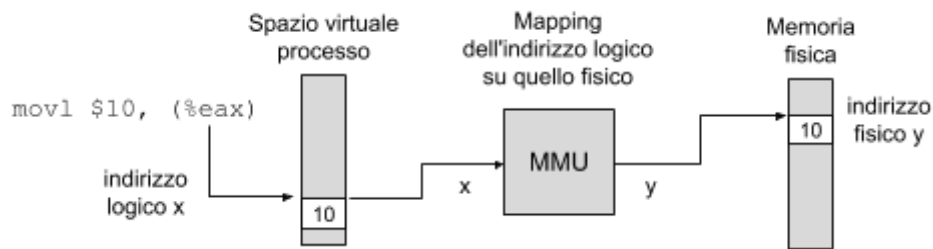
In questo paragrafo descriviamo il funzionamento di un sistema di memoria virtuale e il modo in cui alloca memoria ai processi.

5.1.3.1 Mapping tra indirizzi virtuali e indirizzi fisici: MMU

La memoria virtuale si basa sulla presenza di un **memory management unit** (MMU), cioè un modulo hardware che **mappa** (ovvero traduce) indirizzi nello spazio virtuale (o logico) di un processo su indirizzi della memoria fisica. Gli **indirizzi noti a un programma** (es. puntatori) saranno sempre indirizzi **virtuali** (o logici) e verranno tradotti in indirizzi fisici dal MMU al momento dell'accesso a memoria, in maniera del tutto trasparente al programma.

²⁷ Si pensi ad esempio a malware o virus.

Esempio. La seguente figura si riferisce a un sistema con **memoria virtuale** e mostra la traduzione da parte del MMU dell'indirizzo logico x contenuto nel registro `eax` in un indirizzo fisico y al momento dell'esecuzione di un'istruzione `movl $10, (%eax)`:



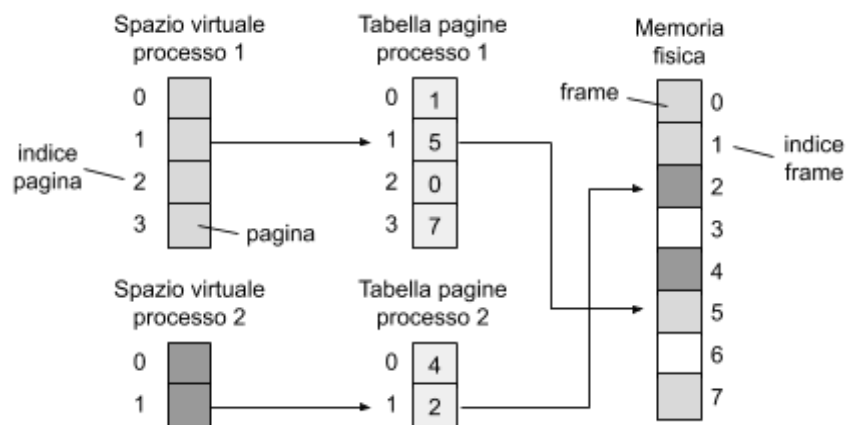
Si noti che il programma che esegue la `mov` non ha alcuna visibilità dell'indirizzo fisico y corrispondente a quello virtuale x.

5.1.3.2 Paginazione

Una delle tecniche più usate per realizzare un mapping tra indirizzi logici e indirizzi fisici in un sistema di memoria virtuale è la **paginazione**, che si basa sui seguenti concetti.

- Lo **spazio logico** di memoria di ciascun processo viene suddiviso implicitamente in blocchi della stessa dimensione chiamati **pagine**, tipicamente di 4 KB oppure 8 KB. Ogni pagina è identificata da un **indice di pagina** che parte da zero per ciascun processo.
- Lo **spazio fisico** di memoria del sistema viene suddiviso implicitamente in blocchi chiamati **frame**, della stessa dimensione di una pagina. Ogni frame è identificato da un **indice di frame** che parte da zero.
- Un array accessibile al MMU chiamato **tabella delle pagine** stabilisce per ogni indice di pagina qual è l'indice del frame che contiene i dati memorizzati nella pagina. Ogni processo ha la **propria** tabella delle pagine.

Esempio. La figura seguente illustra come una memoria fisica di 8 frame viene allocata a due processi con spazi virtuali composti da 4 e 2 pagine, rispettivamente:



I frame in bianco sono frame liberi oppure frame usati dal sistema operativo per le proprie strutture dati come le tabelle delle pagine stesse.

Si noti come lo spazio assegnato ai processi, che è virtualmente contiguo, in realtà potrebbe non esserlo fisicamente. Questo consente di allocare ai processi spazi di dimensioni diverse **senza generare frammentazione esterna**. Tuttavia, poiché la pagina è la minima unità di blocco allocabile, la paginazione potrebbe generare **frammentazione interna**.

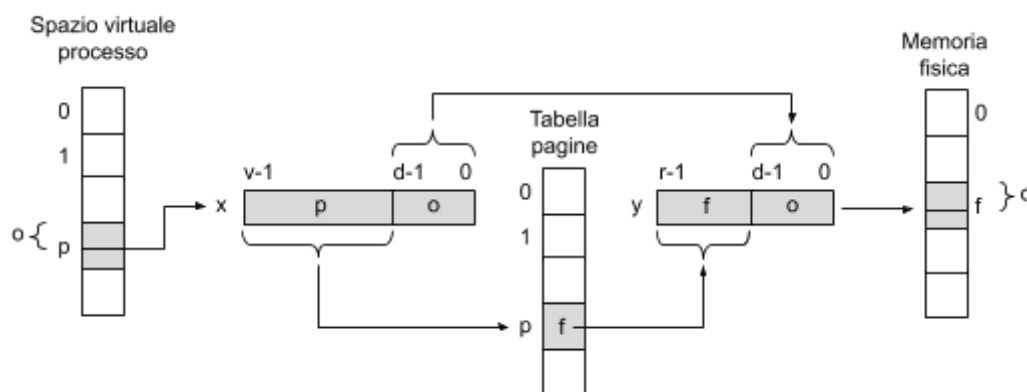
5.1.3.3 Come avviene il mapping degli indirizzi logici su quelli fisici?

Supponiamo di avere uno spazio virtuale di 2^v byte (ad esempio con $v=32$ o $v=64$), pagine di 2^d byte (ad esempio con $d=12$, cioè pagine da 4 KB) e spazio fisico di 2^r byte (ad esempio $r=33$, cioè 8 GB di RAM). Come possiamo tradurre un indirizzo virtuale x a v bit in un indirizzo fisico y a r bit?

Una tecnica comunemente usata è quella di suddividere sia x che y ciascuno in due parti:

- **v-d bit più significativi di x** : indice p della pagina che contiene x ;
- **d bit meno significativi di x** : numero di byte o (**offset**) che separano l'indirizzo x dalla base della pagina p .
- **r-d bit più significativi di y** : indice f del frame che contiene y ;
- **d bit meno significativi di y** : numero di byte o (**offset**) che separano l'indirizzo y dalla base del frame f .

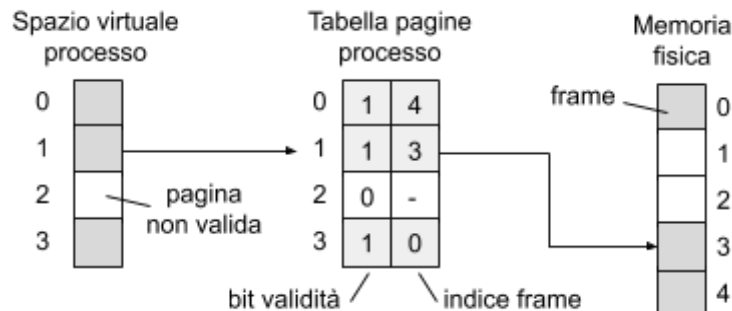
Per costruire y a partire da x , usiamo gli stessi d bit meno significativi (offset) e rimpiazziamo i rimanenti $v-d$ bit con gli $r-d$ bit ottenuti consultando la entry di indice p della tabella delle pagine come illustrato nella seguente figura:



5.1.3.4 Paginazione con bit di validità

Nell'esempio visto sopra, l'intero spazio virtuale è allocato nella memoria fisica, il che potrebbe essere troppo oneroso. Nella pratica invece, lo **spazio logico di un processo coincide generalmente con quello indirizzabile** da un puntatore. Ad esempio, su una macchina con indirizzi a 32 bit, lo spazio logico indirizzabile da un processo è di 2^{32} byte (4 GB ~ 4 miliardi di byte), mentre su una piattaforma a 64 bit è di 2^{64} (16 EB = 16 exabyte ~ 16 trilioni di byte).

Chiaramente, in uno schema del genere la somma delle dimensioni degli spazi di indirizzamento dei processi potrebbe ovviamente eccedere la disponibilità di memoria fisica. Per risolvere questo problema, si usa una tabella delle pagine con **bit di validità**, che vale 1 se alla pagina corrisponde un frame fisico, e 0 altrimenti, come mostrato nella seguente figura:

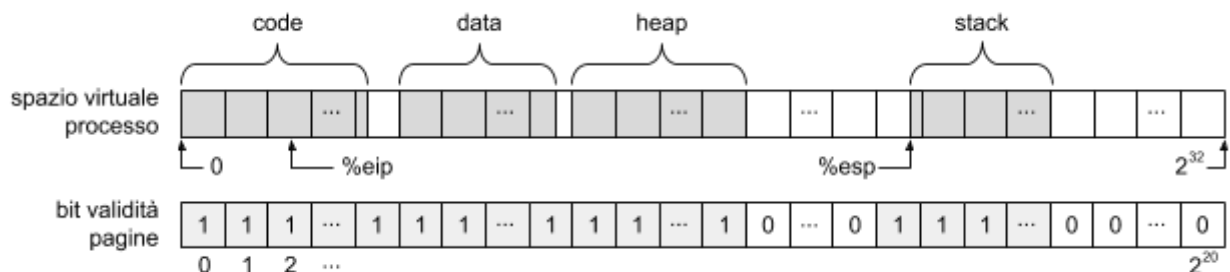


Se un processo tenta di accedere a un indirizzo che ricade in una pagina con il bit di validità a 0, il MMU genera un interrupt (**general protection error**, oppure **segmentation fault** su alcuni sistemi).

La paginazione con bit di validità ha vari vantaggi:

1. **Tutti i processi hanno uno spazio virtuale della stessa dimensione** che coincide con l'intero spazio indirizzabile da un puntatore del programma, semplificando la gestione della memoria.
2. E' possibile **mappare selettivamente sulla memoria fisica le zone effettivamente in uso da un processo** (code, data, heap, stack), senza che lo spazio inutilizzato fra di esse occupi spazio fisico.

Esempio. La seguente figura mostra lo spazio logico tipico di un processo a 32 bit con pagine da 4 KB e i corrispondenti bit di validità delle pagine. Le pagine hanno indici compresi tra 0 e $2^{20}-1$.



Si noti la presenza di frammentazione interna nelle pagine allocate alla fine delle varie sezioni dovute al fatto che non sono sfruttate per intero. Si osservi inoltre come le pagine tra heap e stack e dopo la stack non siano valide, e quindi non occupino spazio di memoria fisica.

5.1.4 Allocazione nella memoria logica: malloc e free

Il livello di allocazione a diretto contatto con le applicazioni lavora interamente in spazio utente e si basa in C sulle primitive `malloc` e `free`²⁸. Queste primitive partizionano lo spazio heap, controllato principalmente mediante la system call `sbrk`²⁹, in:

1. spazio allocato con `malloc` e in uso all'applicazione;
2. spazio libero, potenzialmente allocabile;
3. spazio che contiene informazioni utili all'allocatore per il suo funzionamento.

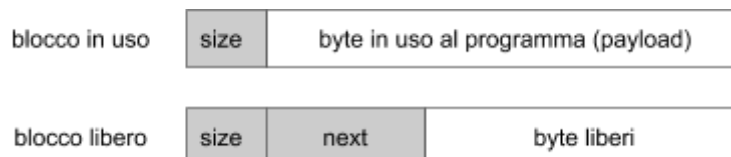
Vi sono molti modi di gestire un allocatore di memoria e in questo paragrafo vedremo uno dei più semplici. Il concetto su cui ruota è quello di una regione contigua dell'heap chiamata **blocco**, che può essere **in uso** se include spazio allocato all'applicazione o **libero** altrimenti. In questa visione, l'heap è partizionato in blocchi liberi o in uso. Questo significa che:

1. l'intero heap è suddiviso in blocchi e non c'è alcuna regione dell'heap che non ricada in qualche blocco;
2. non c'è alcuna intersezione tra blocchi, vale a dire ogni byte dell'heap ricade a un solo blocco.

Ogni blocco ha una **header** che contiene informazioni utili all'allocatore:

1. *Header di un blocco in uso*: contiene la dimensione del blocco (block size);
2. *Header di un blocco libero*: contiene la dimensione del blocco (block size), seguita da un puntatore al blocco libero successivo.

L'immagine seguente illustra il caso di size a 32 bit e puntatori a 64 bit:

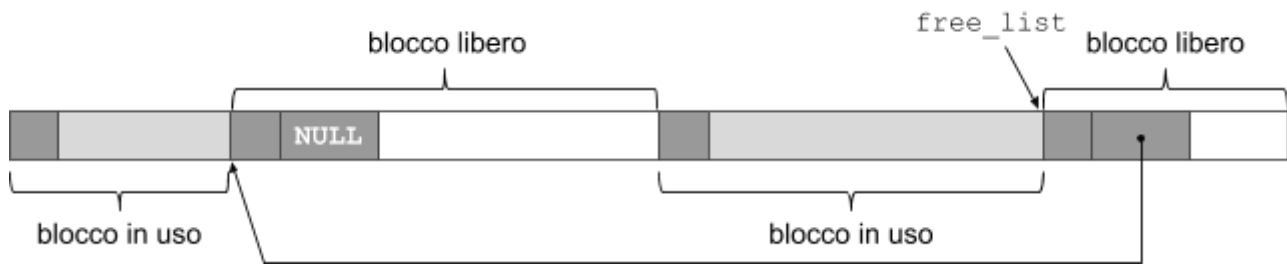


La zona di spazio allocato a un programma è detta payload. I **blocchi liberi** sono pertanto organizzati in una **lista semplice**, dove i nodi della lista sono i blocchi liberi stessi e il campo `next`, che punta al successivo blocco libero, è ricavato dallo spazio che sarebbe in uso al programma per un blocco allocato. Si noti che la minima dimensione di un blocco è di 4 (campo `size`) + 8 (campo `next`) = 12 byte, il che implica che il payload minimo è di 8 byte. Allocando quindi meno di 8 byte otterremmo comunque l'allocazione di 8 byte di payload, più il campo `size`.

Una variabile globale `free_list` punta al primo blocco libero della lista, e vale `NULL` se non vi sono blocchi liberi. La figura seguente illustra una possibile configurazione dell'heap:

²⁸ Ignoriamo per semplicità `calloc` e `realloc`.

²⁹ In alcuni ambienti la system call è deprecata a favore di `mmap`.



Le operazioni di `malloc` e `free` possono essere schematizzate come segue:

- `p=malloc(n)`: cerca nella lista `free_list` un blocco libero di dimensioni sufficienti a contenere il payload `n` richiesto e lo toglie dalla lista dei blocchi liberi. La scelta può essere fatta in vari modi, fra cui:
 - a. *first-fit*: viene scelto il primo blocco libero incontrato;
 - b. *best-fit*: viene scelto il più piccolo blocco libero sufficiente a contenere il payload richiesto.

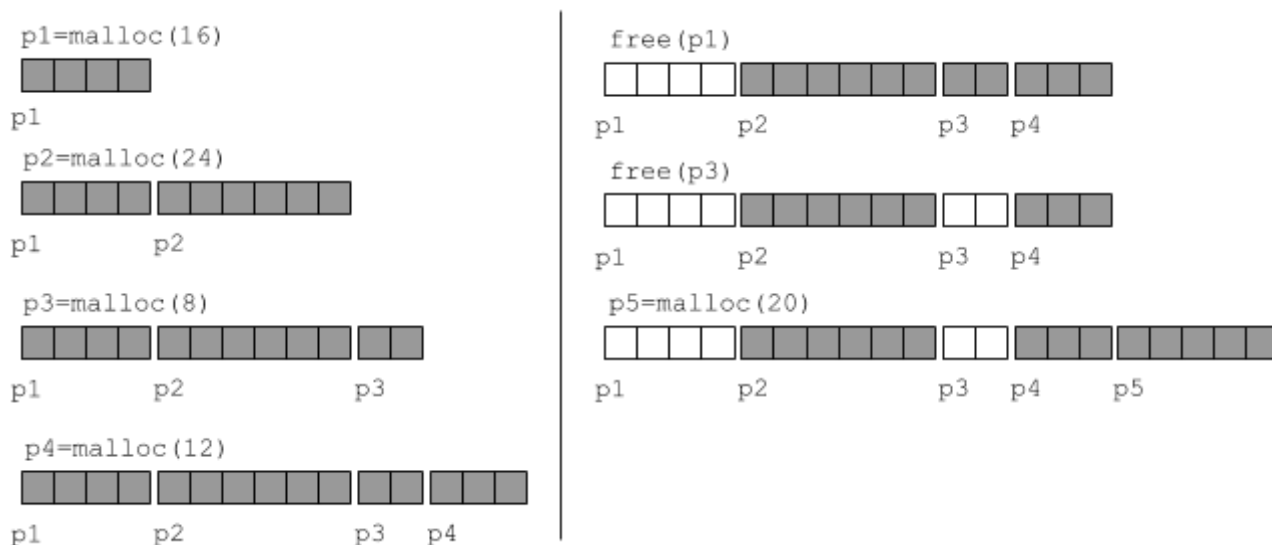
Se non vi sono blocchi liberi sufficientemente grandi, espande la stack di `n+4` byte usando la system call `sbrk(n+4)` e crea un nuovo blocco in uso nello spazio così creato. Infine, restituisce l'indirizzo `p` del primo byte del payload del blocco messo in uso.

- `free(p)`: aggiunge il blocco che inizia all'indirizzo `p-4` in testa alla lista puntata da `free_list`.

Miglioramenti: vi sono numerosi possibili miglioramenti allo schema base proposto, fra cui:

- *Fusione di blocchi liberi*: quando ci sono blocchi adiacenti liberi, possono essere fusi in un blocco unico. Questo richiede alcune modifiche allo schema base come mantenere un bit che marca i blocchi liberi da quelli in uso.
- *Partizione di blocchi liberi*: se un blocco libero viene allocato solo in parte, la rimanenza, se sufficientemente grande, viene convertita in un blocco libero.
- *Liste di blocchi liberi di dimensioni diverse*: invece di avere una sola lista di blocchi liberi, si mantengono liste multiple che permettono di identificare più rapidamente un blocco libero di dimensioni adeguate per un'allocazione.
- *Allineamento*: poiché gli accessi a memoria allineati a indirizzi multipli della dimensione dell'oggetto acceduto garantiscono migliori prestazioni, le `malloc` degli allocatori solitamente restituiscono indirizzi multipli di 4, 8, o addirittura 16 byte.

Esempio: per comprendere come può variare la configurazione dell'heap a fronte di una sequenza di `malloc` e `free`, consideriamo un esempio in cui volutamente ignoriamo la presenza delle header e ci concentriamo sulla struttura generale. Nel nostro esempio, consideriamo con un quadratino una regione di 4 byte, usando il grigio per indicare spazio in uso all'applicazione (allocato) e il bianco per indicare spazio libero. Per semplicità, allochiamo i byte a multipli di 4 byte.



Si noti la presenza di frammentazione esterna: l'ultima allocazione espande l'heap di 20 byte anche se la somma dei byte liberi, presi non consecutivamente, è maggiore. Alla fine della sequenza di operazioni, la dimensione totale dell'heap è di 80 byte, di cui 24 non in uso. Una caratteristica degli allocatori di memoria come `malloc/free` è che non è possibile compattare lo spazio spostando blocchi in memoria: l'applicazione infatti non troverebbe gli stessi dati agli indirizzi restituiti da `malloc`.

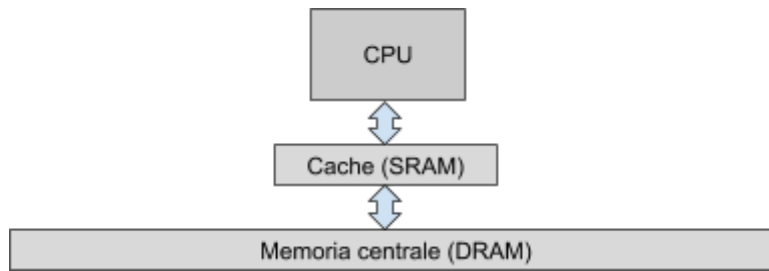
5.2 Come ottenere le migliori prestazioni usando la memoria?

Finora abbiamo considerato la memoria centrale come un grosso array con una visione piatta dove un accesso a memoria ha le stesse prestazioni indipendentemente dall'indirizzo acceduto. Se fosse così, accedere a un indirizzo in RAM costerebbe quasi due ordini di grandezza più che accedere a un registro. Leggere o scrivere un registro richiede infatti tempo dell'ordine di frazioni di nanosecondo, mentre leggere o scrivere in una comune DRAM, la tecnologia più usata per le memorie centrali, richiede circa 100 nanosecondi³⁰. In altre parole, un'apparentemente innocua istruzione `movl $1, (%eax)` costerebbe in termini di tempo 100 volte più che `movl $1, %eax`.

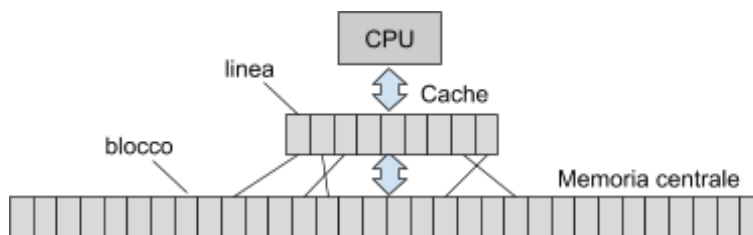
5.2.1 Memorie cache

Per ovviare al problema esposto sopra, i costruttori di hardware hanno pensato fin dagli anni '80 di interporre tra la CPU e la memoria DRAM un'altra memoria, più piccola, costosa e veloce, chiamata **memoria cache**. Una memoria cache è pensata per tenere una copia dei dati più frequentemente acceduti dalla CPU senza dover ogni volta accedere alla DRAM. La memoria cache è costruita tipicamente con tecnologia SRAM (static RAM), più costosa ma anche sostanzialmente più veloce nei tempi di accesso di una DRAM (dynamic RAM).

³⁰ https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html



Una cache è costituita da un certo numero di **linee di dimensione fissa** (es. 64 byte) che possono contenere copie di dati della memoria DRAM. Si assume che la DRAM sia partizionata in blocchi della stessa dimensione della linea di cache, allineati a indirizzi multipli della dimensione della linea di cache.



Ad ogni istante, solo un numero limitato di blocchi può risiedere nelle linee di cache. Indicizzando i blocchi a partire da zero, dato un qualsiasi indirizzo x , il blocco a cui appartiene si ottiene semplicemente come x/L (divisione intera), dove L è la dimensione in byte della linea di cache. Quando la CPU chiede al sistema di memoria di accedere a un indirizzo x , si verifica se l'indirizzo ricade in un blocco attualmente in cache. Se questo avviene, si ha un **cache hit** e la CPU legge/scrive direttamente nella linea di cache senza dover accedere alla DRAM. Questo è lo scenario più favorevole. In caso contrario, si ha un **cache miss**. In caso di cache miss, è necessario caricare dalla DRAM il blocco che contiene l'indirizzo x e portarlo in cache usando una linea ancora non in uso. Se la cache è piena, è necessario identificare una linea **vittima** e **rimpiazzarla** con il blocco caricato. Se la linea vittima è stata scritta mentre era in cache è anche necessario copiarla all'indietro nella memoria per mantenere consistenza tra cache e memoria e non perdere dati. Come si vede, i cache miss sono eventi non desiderabili con un considerevole dispendio di tempo per essere completati. Si noti che i trasferimenti di dati tra cache e memoria avvengono a unità della dimensione della linea.

5.2.2 Politiche di rimpiazzo delle linee

Vi sono diverse politiche possibili per scegliere la linea vittima. La scelta **ottima** sarebbe quella di sacrificare la linea che **non verrà più acceduta per più tempo**, ma non è possibile conoscere quali accessi a memoria verranno fatti in futuro dalla CPU. Una delle politiche di rimpiazzo più popolari è la **LRU** (Least Recently Used) che sceglie come vittima la linea che **non è stata acceduta da più tempo**. L'assunzione è che se non è stata usata da molto tempo è probabile che non verrà più usata a breve. LRU è quindi un'approssimazione della politica ottima usando il passato per stimare il futuro.

5.2.3 Località spaziale e temporale

Non è difficile convincersi che, se la CPU emettesse richieste di accesso a memoria a indirizzi casuali, la probabilità di accedere a un blocco in cache sarebbe molto bassa, generando continui cache miss. Fortunatamente, i programmi reali non generano accessi casuali, ma tendono a esibire due proprietà dette località spaziale e località temporale:

- **Località temporale:** se un programma accede a un indirizzo, è probabile che accederà nuovamente a quell'indirizzo nel vicino futuro. Si pensi ad esempio a una variabile usata da una funzione.
- **Località spaziale:** se un programma accede a un indirizzo, è probabile che accederà a indirizzi contigui in memoria. Si pensi ad esempio alla scansione sequenziale di un array.

Come detto, queste due principi tendono ad emergere in modo spontaneo nei programmi e sono la motivazione per l'uso delle cache. Queste conterranno infatti i blocchi di memoria correntemente in uso al programma eseguito (località temporale) e garantiranno che, se un indirizzo è in cache, anche quelli vicini ricadranno con buona probabilità nella stessa linea (località spaziale). Purtroppo però non sempre i programmi esibiscono località in modo ottimale ed è il compito del programmatore riformularli per migliorare l'uso della cache.

5.2.4 Esempi

5.2.4.1 Somma di matrice

Per convincerci dell'effetto che può avere un uso non ottimale della cache, consideriamo il semplice compito di sommare gli elementi di una matrice:

Somma per righe	Somma per colonne
<pre>int sum(int** m, int n) { int i, j, s = 0; for (i=0; i<n; i++) for (j=0; j<n; j++) s = s + m[i][j]; return s; }</pre>	<pre>int sum(int** m, int n) { int i, j, s = 0; for (i=0; i<n; i++) for (j=0; j<n; j++) s = s + m[j][i]; return s; }</pre>
Tempo richiesto: 0.273 secondi	Tempo richiesto: 1.406 secondi

Un semplice esperimento su un moderno calcolatore (processore Intel i7 a 2.8 GHz con linee di cache da 64 byte) per matrici sufficientemente grandi rivela prestazioni nettamente peggiori nella versione che scorre la matrice per colonna. Il motivo è che nella scansione per colonna si ha un cache miss ad ogni iterazione del ciclo for interno. Viceversa, la versione che scorre la matrice per righe esibisce località spaziale e genera un cache miss ogni $64/\text{sizeof}(\text{int}) = 64/4 = 16$ iterazioni. Infatti, dopo un cache miss iniziale, i successivi 15 interi saranno in cache. Come risultato,

scorrere la matrice in modo "cache-friendly" porta nel nostro esperimento a uno speedup di $1.406/0.273 = 5.150x$.

Analisi scansione per riga. Consideriamo la sequenza di indirizzi acceduti su una riga generica che assumiamo per semplicità partire dall'indirizzo 0:

00 04 08 12 16 20 24 28 32 36 40 44 48 52 56 60 | 64 68 72 ...

La corrispondente sequenza di blocchi, assunti da 64 byte ciascuno è:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 1 1 1 ...

Assumendo di avere anche una sola linea di cache, avremo un cache miss al primo accesso (in grassetto), che caricherà in cache tutti gli interi con indirizzi da 00 a 60. I successivi 15 accessi (indirizzi da 04 a 60) ricadranno nella stessa linea (località spaziale) e pertanto genereranno cache hit. Il successivo cache miss si avrà accedendo all'indirizzo 64 (blocco 1), per cui possiamo ripetere il ragionamento.

Analisi scansione per colonna. Supponiamo che la matrice abbia anche solo 16 colonne. Assumendo sempre di partire dall'indirizzo 0, la sequenza di indirizzi acceduti sulla prima colonna sarà:

00 16 32 64 80 96 ...

che corrisponde ad accedere alla sequenza di blocchi:

00 01 02 03 04 05 ...

Poiché ogni accesso ricadrà in un blocco diverso, avremo un cache miss ad ogni accesso. Si noti che ogni miss carica un intero blocco in cache, di cui si accede solo al primo elemento, mentre gli altri 15 sono stati caricati inutilmente.

5.2.4.2 Analisi di una sequenza di accessi

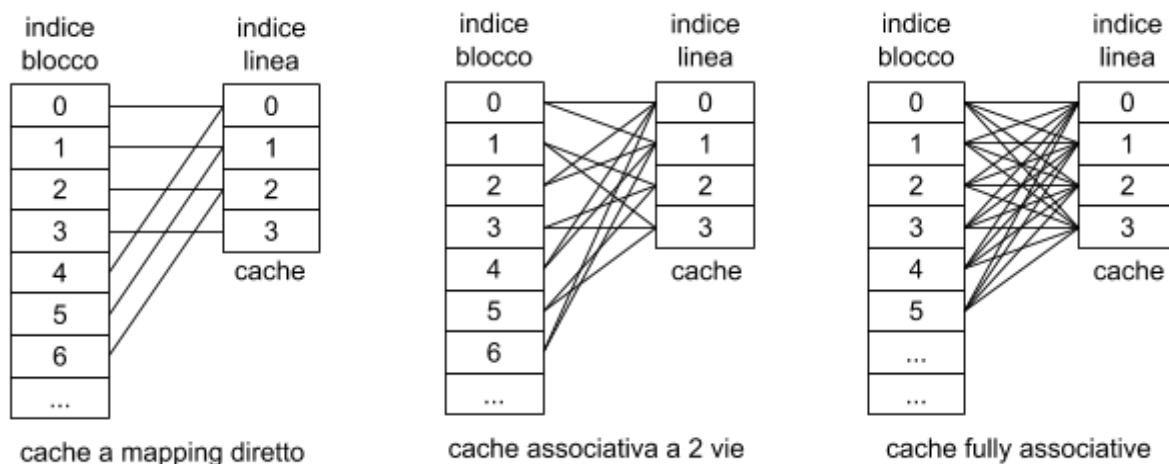
Per illustrare il funzionamento di una cache a fronte di una sequenza di accessi a memoria, consideriamo il seguente esempio che usa una piccola cache con 4 linee da 64 byte ciascuna.

x	20	464	128	36	608	360	204	112	264	268	48	592											
L	0	7	2	0	9	5	3	1	4	4	0	9											
0	0	1	0	2	0	0	0	1	0	2	0	3	0	0	1	1	1	2	1	3	1	4	1
		0	7	1	7	2	7	3	7	0	5	1	5	2	5	3	5	4	5	0	0	1	0
				0	2	1	2	2	2	3	2	0	3	1	3	2	3	3	3	4	3	0	9
					0	9	1	9	2	9	3	9	0	4	0	4	1	4	2	4			

La figura mostra una sequenza di indirizzi (x), gli indici dei blocchi che li contengono (L), e il contenuto della cache dopo ciascun accesso, evidenziano in grassetto la linea appena rimpiazzata. A fianco della cache riportiamo i tempi di permanenza dei blocchi in cache. Come si può vedere, viene sempre rimpiazzata la linea con il contenuto più vecchio come prescritto dalla politica di rimpiazzo LRU. Come si vede, su 12 accessi 10 generano cache miss e si hanno solo 2 cache hit.

5.2.5 Associatività

Negli esempi visti sopra abbiamo assunto che qualsiasi blocco di memoria possa essere caricato in qualsiasi linea di cache. Questo tipo di cache viene detto completamente associativo (fully associative). In realtà le cache reali hanno dei vincoli architetturali che limitano il numero di diverse linee in cui un blocco può essere caricato. Le cache più semplici hanno un mapping diretto, per cui ogni blocco ha un'unica linea in cui può risiedere. Una cache tipica ha invece un numero fisso k di linee in cui un blocco può essere caricato e viene detta associativa a k vie. Valori tipici di k variano tra 2 e 64 anche a seconda della dimensione della cache. La figura seguente illustra le possibili linee di cache in cui può essere caricato un blocco di memoria per i vari tipi di cache discussi sopra.



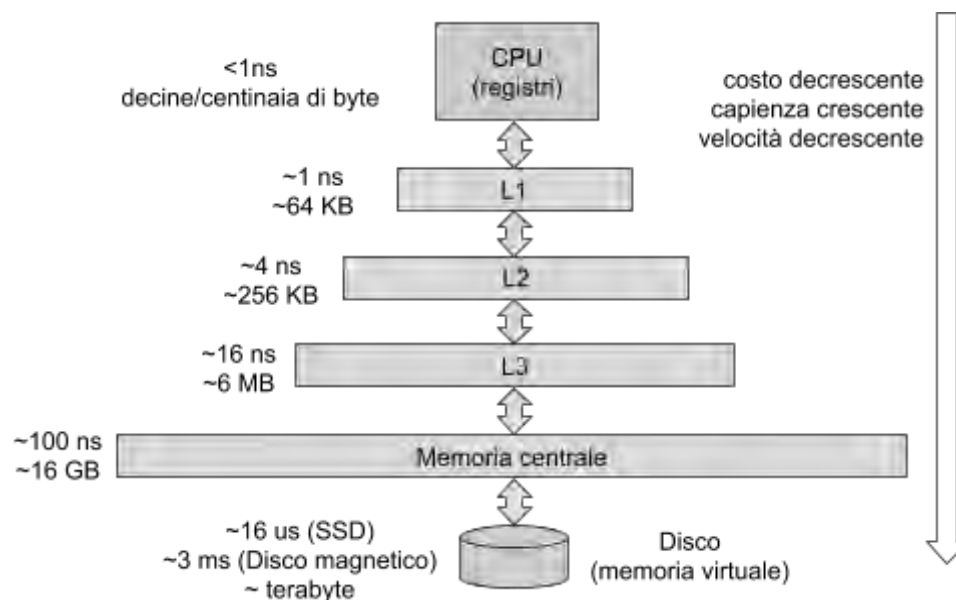
5.2.6 Tipi di cache miss

I cache miss possono essere classificati in tre categorie:

1. **cold** (o **compulsory**): si ha quando un blocco di memoria viene caricato per la prima volta in una linea di cache vuota;
2. **capacity**: si ha quando tutte le linee di cache sono occupate ed è necessario un rimpiazzo di blocco;
3. **conflict**: si ha quando due blocchi di memoria sono mappati sulla stessa linea per vincoli di associatività e non è possibile tenerli entrambi in cache.

5.2.7 Gerarchie di memoria

I sistemi di memoria moderni usano più livelli di cache, dove ogni cache si comporta come descritto nel paragrafo precedente, interfacciandosi verso una memoria più lenta. Queste cache sono chiamate L1, L2, L3, ecc., dove L1 è la cache più piccola e veloce, direttamente acceduta dalla CPU, e L3 quella più grande e lenta.



Nella figura includiamo anche il disco, che viene comunemente utilizzato per tenere i frame di memoria non in uso in un sistema di memoria virtuale (swapped out). Si noti come i livelli più bassi della gerarchia hanno il costo minore per byte, hanno le dimensioni maggiori, ma sono i più lenti. I tempi di accesso riportati sono approssimati, puramente indicativi, e potrebbero variare a seconda delle impostazioni del sistema. Le dimensioni delle memorie sono anch'esse indicative di un PC convenzionale. Il messaggio principale è che le differenze prestazionali si estendono fino a 6 ordini di grandezza: si passa da meno di un nanosecondo per accedere a un registro a 16 microsecondi per un disco SSD o addirittura 3 millisecondi per un disco magnetico convenzionale. In altre parole, un'istruzione `mov` che accede a un dato parcheggiato su disco dalla memoria virtuale può costare un milione di volte in più rispetto all'accesso a un registro. Durante questo tempo la CPU è in **stallo**, in attesa del completamento dell'accesso alla gerarchia di memoria. Nel tempo di attesa potrebbe eseguire milioni di istruzioni. Per questo motivo è essenziale che i programmi esibiscano località e questo richiede attenzione da parte del programmatore.

In un sistema reale come un moderno Intel Core i7, che ha più core indipendenti, le cache L1 ed L2 sono replicate su ciascun core, mentre la cache L3 è condivisa tra i core.

6 Come vengono ottimizzati i programmi?

Idealmente, l'esecuzione di un programma dovrebbe richiedere la minima quantità possibile di risorse di calcolo: in particolare, dovrebbe tenere impegnata la CPU il meno possibile usando allo stesso tempo meno memoria possibile³¹.

Ottimizzare un programma da un punto di vista **prestazionale** significa modificarlo in modo che:

1. rimanga **corretto**: ottimizzazioni che alterino la semantica del programma introducendo errori non sono ammissibili;
2. richieda una **ridotta quantità di risorse di calcolo**: ad esempio comporti l'esecuzione di meno istruzioni o di istruzioni più veloci.

L'ottimizzazione avviene tipicamente a due livelli:

1. **codice**: a parità di algoritmo soggiacente usato, un programma può essere reso più efficiente modificandone la struttura e le istruzioni utilizzate;
2. **algoritmo**: la scelta dell'algoritmo alla base di un programma è cruciale e può fornire miglioramenti prestazionali molto più significativi di qualsiasi ottimizzazione che interviene a livello di codice.

Un'ottimizzazione può essere effettuata dal compilatore o dal programmatore:

1. **compilatore**: i compilatori moderni applicano automaticamente un grande numero di trasformazioni del codice scritto dal programmatore volti ad ottimizzare le risorse richieste dal programma. Ad esempio, compilando un programma in `gcc` con l'opzione `-O1` si attiva un primo livello di ottimizzazioni. Il codice generato è normalmente molto più efficiente rispetto a quello che si otterrebbe compilando senza l'opzione `"-O"`.
2. **programmatore**: non sempre il compilatore riesce a ottimizzare efficacemente un programma. Ad esempio, la scelta dell'algoritmo usato dal programma è delegata al programmatore. Inoltre, in alcuni casi il compilatore è costretto a fare delle assunzioni conservative per garantire che le trasformazioni applicate non rendano potenzialmente scorretto il programma. Come vedremo, queste assunzioni potrebbero essere in alcuni casi rilassate consentendo ottimizzazioni più efficaci applicate a mano dal programmatore.

Un aspetto di particolare importanza che affrontiamo in questo capitolo è capire quali **ottimizzazioni** sono **effettuate dal compilatore**, e quali invece devono essere **effettuate dal programmatore**. Capire questa differenza è importante perché ci consente di non farci perdere tempo su cose che comunque il compilatore farebbe per noi.

6.1 Quanto è importante ottimizzare le prestazioni?

I programmatori professionisti sono abituati a considerare numerose proprietà del software ancor prima di pensare alle prestazioni. **Correttezza, affidabilità, robustezza, usabilità, verificabilità, manutenibilità, riparabilità, evolvibilità, riusabilità, portabilità, leggibilità e modularità** sono

³¹ Vi sono anche altri tipi di risorse che potrebbero essere di interesse, come l'energia consumata. In questa dispensa tratteremo tuttavia solo ottimizzazioni prestazionali volte a ridurre il tempo di esecuzione e la memoria occupata da codice e programmi.

solo alcune delle qualità più rilevanti per lo sviluppo professionale del software. Rendere migliore un programma sotto molti di questi aspetti si paga spesso in termini di una riduzione delle prestazioni. Ad esempio, modularizzare un programma è fondamentale per renderne le singole parti riusabili, ma potrebbe richiedere l'esecuzione di un numero maggiore di istruzioni dovute a salti fra parti diverse di un programma rispetto a un programma "monolitico" ottimizzato interamente per le prestazioni. Un altro aspetto molto importante è la robustezza: inserire test che verifichino se determinate condizioni sono soddisfatte (ed esempio, se certe precondizioni valgono all'ingresso di una funzione) rende generalmente più robusto un programma consentendo di catturare e gestire situazioni impreviste e bug, ma richiede l'esecuzione di un numero maggiore di istruzioni.

Perché ottimizzare le prestazioni se i programmatori danno maggiore importanza ad altri aspetti? La risposta è semplice: **le prestazioni rappresentano in molti casi la "moneta" con cui è possibile "comprare" altre qualità del software.** Rendendo più efficienti alcune parti di un programma ci si può permettere di renderne altre meno efficienti guadagnando però in altri aspetti molto importanti da un punto di vista qualitativo complessivo.

Per anni, il continuo aumento della frequenza di clock dei microprocessori ha fornito ai programmatori "moneta gratis" per realizzare programmi strutturalmente sempre più complessi e articolati, altamente modularizzati e ingegnerizzati per essere robusti e manutenibili. La maggiore "pesantezza" computazionale del software era compensato dai miglioramenti dell'hardware.

Il 2004 ha rappresentato un anno di svolta in cui l'approccio al miglioramento prestazionale dei processori è cambiato sostanzialmente: invece di progettare processori più veloci, tecnologicamente insostenibile per limitazioni fisiche dovuti a problemi di eccessivo riscaldamento, i costruttori hanno iniziato a produrre processori multi-core. Questo d'altra parte ha riversato sui programmatori la necessità di realizzare programmi "paralleli", in grado cioè di sfruttare più unità di calcolo in parallelo. Un programma non sarebbe stato più automaticamente più veloce passando da una generazione di CPU all'altra, ma avrebbe richiesto al programmatore uno sforzo spesso sostanziale per poter usare al meglio più core. Sebbene sia al di là degli scopi di questa dispensa, osserviamo come programmare efficacemente applicazioni parallele corrette ed efficienti rappresenti una delle sfide attuali più importanti per l'ingegneria del software.

6.2 Tecniche di ottimizzazione delle prestazioni di un programma

Vi sono due modi principali per rendere più veloce un programma:

1. **Ridurre il numero di istruzioni eseguite:** sebbene non abbia necessariamente un impatto diretto sulle prestazioni, ridurre il numero di istruzioni eseguite – e quindi il "lavoro" totale (anche detto "work") eseguito dal programma su un dato input – è la principale e più importante tecnica di ottimizzazione;
2. **Ridurre il tempo richiesto per istruzione:** per motivi legati al modo in cui i sistemi di calcolo sono realizzati, non tutte le istruzioni richiedono lo stesso tempo per essere eseguite. Ad esempio, un'operazione di somma è generalmente più veloce di una divisione.

6.2.1 Livelli di ottimizzazione in gcc

In questo capitolo analizzeremo varie ottimizzazioni effettuate dal compilatore `gcc`, studiando in particolare il codice IA32 generato da `gcc 4.8.2` in Linux Mint³². In `gcc` è possibile specificare il livello di ottimizzazione che si richiede usando l'opzione `-Ox`, dove `x` è il livello di ottimizzazione:

- `-O0`: riduce il tempo richiesto per compilare il programma e fa in modo che il debugging produca i risultati attesi. E' l'opzione predefinita (default).
- `-O1` (oppure `-O`): ottimizza. L'ottimizzazione richiede più tempo e molta più memoria per funzioni di grandi dimensioni.
- `-O2`: ottimizza ulteriormente. A questo livello `gcc` esegue quasi tutte le ottimizzazioni che non aumentano la dimensione del codice generato per ridurre il tempo di esecuzione. Rispetto a `-O`, questa opzione incrementa sia il tempo di compilazione che le prestazioni del codice generato. Tipicamente le librerie standard del linguaggio C (`libc`) sono compilate a questo livello.
- `-O3`: ottimizza ancora di più. Alcune ottimizzazioni potrebbero aumentare la dimensione del codice generato.
- `-Os`: ottimizza per ridurre la dimensione del codice; attiva tutte le ottimizzazioni di livello 2 che non aumentano la dimensione del codice, più altre.

Ciascuno dei livelli 1, 2 e 3 abilita tutte le ottimizzazioni del livello precedente e ne introduce delle altre. E' possibile abilitare o disabilitare singole ottimizzazioni: ad esempio, l'opzione `-fomit-frame-pointer` vista nel Capitolo 4 consente di velocizzare le chiamate a funzione omettendo il codice di gestione del registro `ebp`. Un elenco completo di quali trasformazioni sono abilitate sui vari livelli è disponibile sul sito [GNU](https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/Optimize-Options.html#Optimize-Options)³³. Il passaggio da `-O0` a `-O1` fornisce il miglioramento prestazionale più ampio. Passare da `-O1` a `-O2` o da `-O2` a `-O3` ha generalmente un impatto meno marcato.

6.2.2 Ridurre il numero di istruzioni eseguite

Il modo più efficace per ridurre il numero di istruzioni eseguite è quello di applicare degli algoritmi efficienti. Ad esempio, se un programma deve ordinare n elementi, un algoritmo con costo $O(n \log n)$ sarà certamente più veloce di uno $O(n^2)$ se n è abbastanza grande. La scelta dell'algoritmo giusto può portare a guadagni di prestazioni anche di diversi ordini di grandezza.

In questo paragrafo ci concentreremo su ottimizzazioni del codice e analizzeremo alcune delle tecniche comunemente usate dai compilatori per ottimizzare cicli, funzioni e logica di calcolo. Queste tecniche di ottimizzazione si basano sull'**analisi del testo del programma a tempo di compilazione**.

6.2.2.1 Constant folding

La tecnica del **constant folding** (ripiegamento delle costanti) consiste nel rimpiazzare espressioni con operandi costanti con il risultato dell'espressione, riducendo il numero di istruzioni eseguite.

Esempio.

³² `gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2 (gcc --version)`

³³ <https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/Optimize-Options.html#Optimize-Options>

Mostriamo il codice C equivalente che si otterrebbe applicando manualmente il constant folding a un semplice programma C:

C originale: test-cf.c	Constant propagation (manualmente)
<pre>int f() { return 8+(14/2)*3; }</pre>	<pre>int f() { return 29; }</pre>

Osserviamo che `gcc` applica il constant folding automaticamente già a livello di ottimizzazione `-O1`, per cui non è necessario effettuarlo manualmente nei propri programmi:

<code>gcc -O1 -S -fomit-frame-pointer test-cf.c</code>
<pre>f: movl \$29, %eax ret</pre>

6.2.2.2 Constant propagation

Se a una variabile è assegnato un valore costante, occorrenze successive della variabile possono essere rimpiazzate con quel valore, ottenendo un codice più efficiente. Questa tecnica di ottimizzazione è nota come **constant propagation** (propagazione delle costanti) e viene applicata normalmente insieme al constant folding.

Esempio.

Mostriamo il codice C equivalente che si otterrebbe applicando manualmente constant propagation e constant folding a un semplice programma C:

C originale: test-cp.c	Constant propagation (manualmente)	Constant propagation + constant folding (manualmente)
<pre>int x; int f() { x = 8; return x-2; }</pre>	<pre>int x; int f() { x = 8; return 8-2; }</pre>	<pre>int x; int f() { x = 8; return 6; }</pre>

Anche in questo caso notiamo che `gcc` applica la constant propagation automaticamente insieme al constant folding già a livello di ottimizzazione `-O1`, per cui non è in realtà necessario effettuarla manualmente nei propri programmi.

<code>gcc -O1 -S -fomit-frame-pointer -m32 test-cp.c</code>
<pre>f: movl %8, x movl \$6, %eax</pre>

```
ret
```

Se invece compilassimo senza ottimizzazioni otterremmo:

```
gcc -S -fomit-frame-pointer -m32 test-cp.c
```

```
f: movl $8, x      # x = 8
   movl x, %eax    # calcola x-2 e lo mette in eax
   subl $2, %eax
   ret
```

in cui non vengono effettuati né constant propagation né constant folding.

6.2.2.3 Common subexpression elimination

Espressioni complesse che contengono al loro interno sottoespressioni ripetute possono essere semplificate calcolando separatamente le sottoespressioni comuni e riusandone il valore calcolato. Questa tecnica di ottimizzazione viene chiamata **common subexpression elimination** ed è illustrata nel seguente esempio.

Esempio.

Mostriamo il codice C equivalente che si otterrebbe applicando manualmente la common subexpression elimination a un semplice programma C:

C originale: test-cse.c	Common subexpr. elimination (manualmente)
<pre>int expr(int x, int y) { return (x + y)*(x + y); }</pre>	<pre>int expr(int x, int y) { int z = x + y; return z * z; }</pre>

Si noti che `gcc` applica la common subexpression elimination automaticamente già a livello di ottimizzazione `-O1`, per cui in generale non è necessario effettuarlo manualmente nei propri programmi:

```
gcc -O1 -S -fomit-frame-pointer -m32 test-cse.c
```

```
f:  movl    8(%esp), %eax
   addl    4(%esp), %eax
   imull   %eax, %eax
   ret
```

Se invece compilassimo senza ottimizzazioni otterremmo:

```
gcc -S -fomit-frame-pointer -m32 test-cp.c
```

```
f:  movl    8(%esp), %eax      # calcola x+y e lo mette in ecx
    movl    4(%esp), %edx
    leal    (%edx,%eax), %ecx
    movl    8(%esp), %eax      # calcola di nuovo x+y e lo mette in
eax
    movl    4(%esp), %edx
    addl    %edx, %eax
    imull   %ecx, %eax         # calcola (x+y)*(x+y) e lo mette in eax
    ret
```

Alle volte la common subexpression elimination va fatta però a mano, ad esempio nel caso in cui coinvolga chiamate a funzione il cui codice non è noto al compilatore.

6.2.2.4 Dead code elimination

La **dead code elimination** è un'ottimizzazione che consente di identificare e ignorare porzioni di codice che non possono essere in nessun caso eseguite, generando codice più veloce e compatto.

Esempio.

Mostriamo il codice C equivalente che si otterrebbe applicando manualmente la dead code elimination a un semplice programma C:

C originale: test-dce.c	Dead code elimination (manualmente)
<pre>void f() { if (0) return 10; return 20; }</pre>	<pre>void f() { return 20; }</pre>

In questo caso, l'istruzione `return 10` non può essere mai eseguita essendo in un `if (0)` e viene quindi ignorata. Osserviamo che `gcc` applica la dead code elimination automaticamente anche se nessuna ottimizzazione è specificata, per cui non è necessario effettuarla manualmente nei propri programmi:

<code>gcc -S -fomit-frame-pointer -m32 test-dce.c</code>
<pre>f: movl \$20, %eax ret</pre>

6.2.2.5 Loop-invariant code motion (hoisting)

Fra le varie possibili ottimizzazioni dei cicli, il **loop-invariant code motion** (o **hoisting**) consiste nell'identificare porzioni di codice nel corpo di un ciclo che calcolerebbero a stessa cosa ad ogni iterazione (loop-invariant) e spostarle prima del ciclo (code motion). In questo modo, le istruzioni

spostate verranno eseguite una sola volta prima di entrare nel ciclo e non più ad ogni iterazione, portando in alcuni casi vantaggi prestazionali sostanziali.

Come vedremo nei seguenti esempi, non sempre il loop-invariant code motion può essere eseguito dal compilatore. In alcuni casi, deve essere il programmatore ad occuparsene.

Esempio 1.

In questo primo esempio, il loop-invariant code motion può essere applicato direttamente dal compilatore:

C originale: test1-licm.c	Loop-invariant code motion (manualmente)
<pre>int f(int x, int n) { int s = 1; for (; n>0; n--) s *= 7+x; return s; }</pre>	<pre>int f(int x, int n) { int s = 1, z = 7+x; for (; n>0; n--) s *= z; return s; }</pre>

Nel programma originale, l'espressione $7+x$ viene calcolata ad ogni iterazione del ciclo anche se il risultato è sempre lo stesso. Nella versione ottimizzata, il calcolo dell'espressione viene effettuato una sola volta prima di entrare nel ciclo e salvato in una variabile usata poi nel corpo del ciclo. Vediamo il codice IA32 generato da gcc per il programma originale con livello di ottimizzazione -O1:

gcc -S -fomit-frame-pointer -m32 test1-licm.c
<pre>f: movl 8(%esp), %edx testl %edx, %edx jle .L4 movl \$1, %eax movl 4(%esp), %ecx addl \$7, %ecx # 7+x calcolato prima del ciclo .L3: imull %ecx, %eax # inizio ciclo for subl \$1, %edx jne .L3 # fine ciclo for ret .L4: movl \$1, %eax ret</pre>

In questo caso, il compilatore ha effettuato automaticamente l'ottimizzazione.

Esempio 2.

In questo secondo esempio, il loop-invariant code motion deve essere effettuato dal programmatore:

C originale: test2-licm.c	Loop-invariant code motion (manualm.)
<pre>int has_space(const char* s) { int i; for (i=0; i<strlen(s); i++) if (s[i]==' ') return 1; return 0; }</pre>	<pre>int has_space(const char* s) { int i, n = strlen(s); for (i=0; i<n; i++) if (s[i]==' ') return 1; return 0; }</pre>

Osserviamo che il programma originale richiama la funzione di libreria `strlen` (che calcola il numero di caratteri nella stringa) ad ogni iterazione del ciclo `for`, nonostante la lunghezza della stringa rimanda sempre la stessa. Si noti che il costo della funzione `has_space` è $\Theta(n^2)$, dove n è la lunghezza della stringa `s` passata in ingresso! Questo è a tutti gli effetti un **performance bug**³⁴ in quanto introduce un'inefficienza che non dovrebbe esserci: verificare se una stringa di n caratteri contiene spazio richiede tempo $O(n)$.

Qual è l'ostacolo per un compilatore C nell'applicare l'hoisting in situazioni come queste? Diversamente dal caso dell'Esempio 1, in cui il codice spostato è accessibile al compilatore (che ne può analizzare il comportamento e verificare se è invariante nel ciclo), nel caso di una chiamata a funzione non è in generale possibile sapere a tempo di compilazione quale codice verrà eseguito. E' infatti possibile che il codice della funzione chiamata venga stabilito a tempo di esecuzione durante il caricamento di librerie dinamiche. Se la funzione avesse effetti collaterali, ad esempio mantenesse un contatore del numero di volte che è invocata per scopi di monitoraggio del funzionamento del programma, non sarebbe invariante nel ciclo e non potrebbe essere spostata. **In mancanza di informazioni precise, un compilatore fa la scelta più conservativa: non effettuare un'ottimizzazione che potrebbe modificare la semantica del programma originario.** In questi casi sta al programmatore applicare manualmente, se possibile, l'ottimizzazione.

6.2.2.6 Function inlining

Quando una funzione viene chiamata ripetutamente, ad esempio in un ciclo, e il suo corpo contiene poche istruzioni, può essere vantaggioso applicare il **function inlining**. Questa ottimizzazione consiste nel **rimpiazzare una chiamata a funzione con il corpo della funzione chiamata**. I **vantaggi** principali del funtion inlining sono:

1. si evita di dover eseguire istruzioni per il passaggio dei parametri, l'invocazione, la gestione dello stack frame e il ritorno da funzione;
2. poiché il codice del chiamato è innestato direttamente nel chiamante, è possibile applicare nel chiamante ottimizzazioni globali che normalmente rimangono confinate nei limiti della funzione a cui vengono applicate (es. constant propagation);
3. si migliora la località spaziale del codice, evitando di saltare a parti di codice che non sono in cache e riducendo i cache miss.

³⁴ Bug di questo tipo sono frequenti in applicazioni reali. Ad esempio, un bug del tutto analogo a quello visto nel nostro esempio appare nel programma `wf` per il calcolo delle frequenze delle parole in un testo distribuito in Linux Fedora 17–Beefy Miracle.

Lo **svantaggio** principale del function inlining è un aumento della dimensione del codice (code bloat), anche se generalmente questo non ha un impatto rilevante.

Esempio.

Consideriamo l'effetto dell'inlining di una semplice funzione:

C originale: test-inl.c	Function inlining (manualmente)
<pre>int sqr(int x) { return x*x; } void f(int* v, int n) { int i; for (i=0; i<n; i++) v[i] = sqr(i); }</pre>	<pre>void f(int* v, int n) { int i; for (i=0; i<n; i++) v[i] = i*i; }</pre>

Osserviamo come rimpiazzando l'invocazione della funzione `sqr(i)` con il corpo della funzione stessa si risparmiano numerose istruzioni. Il function inlining viene applicato automaticamente da gcc a livello di ottimizzazione `-O3`, oppure mediante l'opzione `-finline-functions`:

gcc -S -fomit-frame-pointer -m32 test-inl.c
<pre>f: L6: movl %eax, %edx # inizio ciclo for imull %eax, %edx # calcola i*i (non viene chiamata sqr) movl %edx, (%ebx,%eax,4) addl \$1, %eax cmpl %ecx, %eax jne .L6 # fine ciclo for ...</pre>

6.2.3 Ridurre il costo delle istruzioni eseguite

Un secondo approccio per velocizzare un programma consiste nel rimpiazzare istruzioni con alternative più veloci o fare in modo di diminuirne il tempo di esecuzione modificando il contesto in cui vengono eseguite. Vediamo degli esempi nei paragrafi successivi.

6.2.3.1 Register allocation

Poiché i registri della CPU sono le memorie più veloci di un sistema di calcolo, i compilatori cercano di fare in modo che oggetti frequentemente acceduti da un programma siano tenuti in registri. Ad esempio, è ragionevole assumere che una variabile ripetutamente usata nel corpo di

un ciclo sia associata a un registro. In C è possibile suggerire al compilatore che una variabile locale sia tenuta in un registro usando la parola chiave `register` nella sua dichiarazione:

```
void f() {
    register int x; // variabile da tenere se possibile in un
    registro
    ...
}
```

I moderni compilatori tentano di tenere le variabili il più possibile nei registri indipendentemente dal fatto che vi sia o meno il qualificatore `register`. A questo scopo, vengono applicati algoritmi di allocazione dei registri (**register allocation**) che associano un registro o una locazione di memoria ad ogni variabile usata in una data porzione di codice. L'obiettivo è fare in modo che ai registri più frequentemente usati siano associati registri. Poiché i registri sono in numero limitato, non è detto che questo sia possibile: una variabile potrebbe essere associata a un registro in una determinata porzione di codice, ma poi potrebbe essere necessario liberare il registro per altri usi, trasferendone il contenuto in memoria. Questa operazione viene chiamata **register spilling**.

Il compilatore `gcc` effettua l'allocazione automatica dei registri già a livello di ottimizzazione `-O1`.

Esempio.

Vediamo un esempio di compilazione con e senza allocazione dei registri:

C originale: <code>test-ra.c</code>	Senza alloc. registri (<code>-O0</code>)	Con alloc. registri (<code>-O1</code>)
<pre>void f(int n) { while (n--) g(); }</pre>	<pre>f: jmp L3 L4: call g L3: movl 4(%esp), %eax leal -1(%eax), %edx movl %edx, 4(%esp) testl %eax, %eax jne L4 ret</pre>	<pre>f: pushl %ebx movl 8(%esp), %ebx testl %ebx, %ebx je L3 L5: call g subl \$1, %ebx jne L5 L3: popl %ebx ret</pre>

Nel primo caso la variabile `n` è sempre associata alla cella di memoria di indirizzo `esp+4`; ad ogni iterazione nel ciclo vengono effettuati due accessi a memoria (in grassetto): uno in lettura e uno in scrittura. Nel secondo caso, la variabile `n` è caricata una volta all'inizio dall'indirizzo `esp+4` nel registro `ebx`³⁵ e nel ciclo viene tenuta in quel registro, velocizzando l'accesso.

6.2.3.2 Strength reduction

In alcuni casi, è possibile rimpiazzare un'istruzione con una meno costosa in termini di tempo di calcolo. Questa tecnica di ottimizzazione viene chiamata **strength reduction** ed è spesso applicata automaticamente dai compilatori. Ad esempio, si può sfruttare la proprietà che la

³⁵ Ci si potrebbe chiedere: perché il `gcc` usa il registro callee-save e non uno caller-save? Anche questa è un'ottimizzazione: così facendo, non deve salvare/ripristinare `ebx` prima/dopo ogni chiamata a `g` nel ciclo.

moltiplicazione di un intero per una potenza di due è equivalente a uno shift a sinistra, mentre una divisione di un intero per una potenza di due è equivalente a uno shift aritmetico a destra. In alcuni casi, è possibile usare somme e sottrazioni da sole o insieme agli shift. Le operazioni di shift, addizione e sottrazione sono sostanzialmente più veloci di moltiplicazioni e divisioni.

Esempio 1.

Operazione originaria	Operazione equivalente più efficiente
$x / 8$	$x \gg 3$
$x * 64$	$x \ll 6$
$x * 2$	$x \ll 1$ oppure $x + x$
$x * 15$	$(x \ll 4) - x$

La strength reduction è una delle ottimizzazioni più frequentemente applicate dai compilatori. Vediamo un esempio di codice IA32 generato da `gcc`:

C originale: test-sr.c	<code>gcc -S -m32 -O1 -fomit-frame-pointer test-sr.c</code>
<pre>int f(int x) { return 17*x; }</pre>	<pre>f: movl 4(%esp), %eax # eax = x movl %eax, %edx # edx = eax sall \$4, %edx # edx = edx << 4 addl %edx, %eax # eax = eax + edx ret # return eax</pre>

Il calcolo di $17*x$ viene ridotto a $16*x+x$, equivalente a $(x \ll 4) + x$.

Esempio 2.

In quest'altro esempio, mostriamo come ricondurre moltiplicazioni dove entrambi i fattori sono non costanti ad addizioni ripetute:

C originale: test-sr1.c	Strength reduction (manualmente)
<pre>void f(int* v, int n, int k) { int i; for (i=0; i<n; i++) v[i] = i*k; }</pre>	<pre>void f(int* v, int n, int k) { int i, c = 0; for (i=0; i<n; i++) { v[i] = c; c += k; } }</pre>

Questa ottimizzazione viene effettuata automaticamente da `gcc` già a livello `-O1`.

6.2.3.3 Riduzione dei cache miss

Una stessa istruzione con operandi in memoria può richiedere tempi di esecuzione molto diversi a seconda che l'indirizzo acceduto ricada in cache o meno. Uno degli aspetti più cruciali per le prestazioni di un programma è pertanto la **località degli accessi a memoria**. Diversamente dalla maggior parte delle tecniche di ottimizzazione viste nei paragrafi successivi, ottimizzare un programma affinché esibisca maggiore località è tipicamente **compito del programmatore** piuttosto che del compilatore.

Esempio.

Analizziamo un primo semplice caso che consiste nello scorrere una matrice, ad esempio per calcolarne la somma degli elementi:

C originale: test-mat.c	Ottimizzazione località (manualmente)
<pre>int mat_sum(int** m, int n) { int i, j, s = 0; for (i=0; i<n; i++) for (j=0; j<n; j++) s += m[j][i]; return s; }</pre>	<pre>int mat_sum(int** m, int n) { int i, j, s = 0; for (i=0; i<n; i++) for (j=0; j<n; j++) s += m[i][j]; return s; }</pre>

La versione originale scorre la matrice per colonne, la versione ottimizzata invece per righe. L'unica differenza fra i due programmi è infatti nell'ordine degli indici nell'accesso alla matrice *m*. L'accesso per righe esibisce una località spaziale maggiore, poiché dopo ogni cache miss un certo numero di elementi acceduti alle iterazioni successive saranno in cache. Viceversa, l'accesso per colonne implica di saltare ad ogni iterazione a indirizzi spazialmente lontani gli uni dagli altri. La versione ottimizzata può essere oltre 3 volte più efficiente di quella originaria!

6.2.3.4 Allineamento dei dati in memoria

Una regola applicata normalmente dai compilatori e chiamata **allineamento** è che ogni oggetto di *s* byte immagazzinato in memoria inizi a un indirizzo *x* multiplo di *s*, cioè tale che $x \bmod s = 0$. Ad esempio, ogni *short* inizierà ad un indirizzo pari, un *int* a un indirizzo multiplo di 4. La motivazione è che su alcuni processori un accesso a memoria disallineato può comportare una perdita di prestazioni o addirittura un'eccezione. Per garantire l'allineamento, i compilatori riservano dello spazio di memoria fra oggetti consecutivi in memoria che funge da cuscinetto in modo che l'oggetto successivo sia allineato correttamente. Questo spazio di memoria non in uso viene chiamato **padding**. L'uso del padding è un classico modo per barattare prestazioni con spazio di memoria utilizzato (time-space tradeoff): si ottiene codice più veloce che però richiede più spazio.

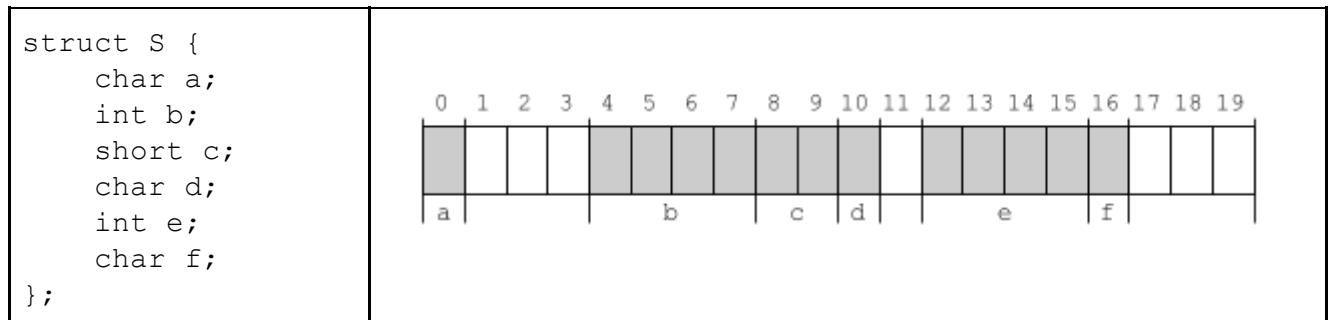
L'allineamento viene effettuato ovunque vengano memorizzati oggetti nello spazio di memoria di un processo:

- **data**: variabili globali (dichiarate in C fuori dal corpo delle funzioni);

- **stack frame:** parametri passati e variabili locali delle funzioni;
- **heap:** ogni oggetto allocato avrà una base multiplo di una potenza di 2, tipicamente 16 byte;
- **codice:** l'indirizzo di una funzione è generalmente allineato a un multiplo della dimensione di una linea di cache (un valore tipico è 64 byte);
- **strutture:** campi di tipi primitivi.

Esempio.

Consideriamo una struttura C con campi di dimensioni diverse:



Per garantire l'allineamento dei singoli campi, essi saranno disposti in memoria come illustrato a destra nella tabella precedente. Le celle in bianco rappresentano spazio di padding inserito dal compilatore per garantire l'allineamento. I numeri in alto rappresentano l'offset di ogni singolo byte all'interno della struttura. Si noti che sia l'indirizzo dell'intera struttura che la sua dimensione dovranno essere multipli della dimensione del suo campo di tipo primitivo più grande. In questo modo, in un array di strutture ogni campo sarà correttamente allineato.

6.2.4 Ridurre lo spazio di memoria

Vi sono vari modi di ottimizzare lo spazio di memoria richiesto da un programma. Essi incidono su:

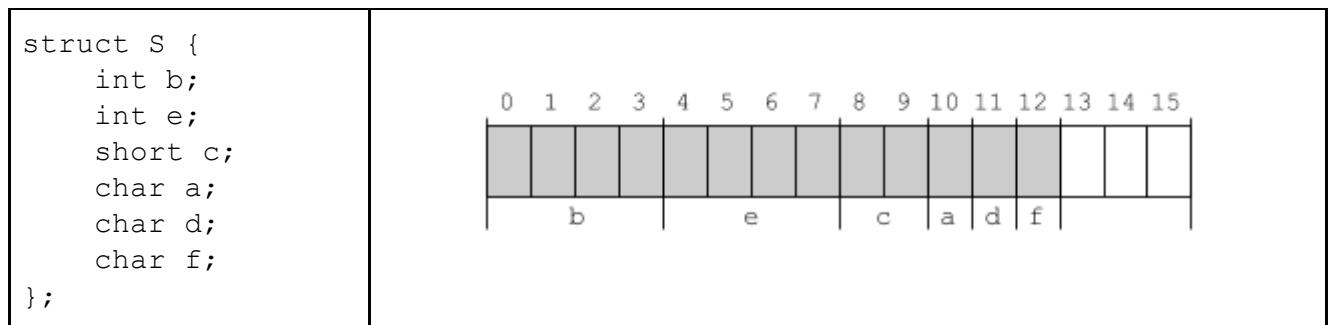
- **Spazio occupato dal codice:** i compilatori possono effettuare numerose ottimizzazioni per ridurre lo spazio di memoria richiesto dal codice un programma. Come abbiamo visto nel [Paragrafo 6.2.1](#), questo può essere fatto ad esempio compilando in `gcc` con l'opzione `-Os`.
- **Spazio occupato dai dati:** conoscendo il modo in cui vengono rappresentate le informazioni in un sistema di calcolo, il programmatore può selezionare la rappresentazione più adatta. Ad esempio, l'uso di array è generalmente meno dispendioso di memoria rispetto alle strutture collegate.

6.2.4.1 Ottimizzare lo spazio richiesto dalle strutture C

Come abbiamo visto nel [Paragrafo 6.2.3.4](#), i campi delle strutture includono normalmente padding per garantire l'allineamento dei campi. Per minimizzare lo spazio richiesto da una struttura è possibile semplicemente cambiare l'ordine in cui appaiono i campi nella dichiarazione della struttura in modo che **i campi più grandi appaiano per primi**.

Esempio.

Consideriamo nuovamente la struttura C dell'esempio del [Paragrafo 6.2.3.4](#) e mostriamo cosa si ottiene riordinando le dichiarazioni dei campi:



Osserviamo che dopo l'ottimizzazione la dimensione complessiva della struttura è passata da 20 a 16 byte, riducendosi del 20%.

6.2.5 Le ottimizzazioni dei compilatori sono le migliori possibili?

Sebbene i **compilatori** applichino algoritmi di ottimizzazione sofisticati, **non sempre riescono a ottenere un risultato ottimo**. Ad esempio, l'allocazione dei registri è in generale un problema NP-completo [Chaitin et al. 1981] e quindi vengono applicati algoritmi approssimati.

Alle volte analizzando il codice assembly generato si possono identificare opportunità di **ottimizzazioni che possono essere applicate manualmente al codice assembly**, ad esempio generando un file `.s` con `gcc -S` e poi editandolo a mano. Osserviamo che questa operazione richiede **molta cautela**:

1. spesso quello che pensiamo possa essere un'ottimizzazione ha poco impatto sulle prestazioni e potrebbe addirittura peggiorarle: ha senso intervenire solo se le istruzioni che si modificano sono effettivamente un collo di bottiglia prestazionale (vedi Paragrafo 5.3).
2. il rischio di introdurre errori nel programma è molto alto.

6.3 Quali parti di un programma ottimizzare?

Per ottimizzare efficacemente un programma è necessario innanzitutto identificare se ci sono parti critiche per le prestazioni e quali sono. Il programmatore deve:

1. Stimare il **miglioramento complessivo** nelle prestazioni del programma che si otterrebbe **ottimizzando le singole parti**. Questo ci aiuta a capire il potenziale impatto di un'ottimizzazione e capire se vale la pena di applicarla.
2. Identificare le porzioni di codice che consumano la maggior parte del tempo di esecuzione o dello spazio di memoria usato da un programma. Queste porzioni di codice vengono chiamate **hot spot** o **colli di bottiglia**. A questo scopo, è possibile usare programmi di analisi delle prestazioni chiamati **performance profiler**.

6.3.1 Scala degli eventi in un sistema di calcolo

Per comprendere dove è più probabile che emergano i colli di bottiglia in un sistema di calcolo, e quindi sia opportuno focalizzare le ottimizzazioni, è fondamentale avere un'intuizione del tempo relativo richiesto da varie operazioni.

Evento ³⁶	Latenza effettiva	Latenza riscalata
Ciclo di clock	0.4 nsec	1 sec
Accesso cache L1	0.9 nsec	2 sec
Accesso cache L2	2.8 nsec	7 sec
Accesso cache L3	28 nsec	1 min
Accesso RAM DDR3 DIMM	100 nsec	4 min
Accesso disco SSD	50-150 μ sec	1.5-4 giorni
Accesso disco a rotazione	1-10 msec	1-9 mesi
Invio pacchetto Internet continentale/intercontinentale	65-141 msec	5-11 anni

6.3.2 Speedup e legge di Amdahl

La legge di Amdahl³⁷ ci permette di stimare l'impatto prestazionale che l'ottimizzazione di una parte di un programma ha sul tempo di esecuzione del programma nel suo insieme.

Speedup. Sia T il tempo di esecuzione di un programma e sia T' il tempo di esecuzione dopo aver applicato un'ottimizzazione. Il rapporto $S = \frac{T}{T'}$, viene chiamato **speedup** e caratterizza il miglioramento prestazionale dovuto all'ottimizzazione.

Lo speedup è un numero adimensionale dato dal rapporto tra misure prestazionali. Si noti che se $S > 1$ il programma ottimizzato è più veloce di quello originario.

Esempio.

Supponiamo che un programma richieda $T = 10$ secondi su determinati dati di input. Dopo aver ottimizzato il programma, il tempo di esecuzione sugli stessi dati di input diventa $T' = 7$ secondi. Lo speedup dovuto all'ottimizzazione è $S = T/T' = 10/7 = 1.42x$. Per indicare che il programma ottimizzato è 1.42 volte più veloce di quello non ottimizzato usiamo la notazione 1.42x.

³⁶ <https://www.prowesscorp.com/computer-latency-at-a-human-scale/>

³⁷ Sebbene sia generalmente formulata per stimare l'effetto sulle prestazioni dovuto all'uso di più processori per effettuare calcoli in parallelo, ne riportiamo qui una variante pensata per programmi sequenziali che non prevedono calcoli paralleli.

Legge di Amdahl. Supponiamo di dividere un programma in due parti A e B . Sia T il tempo totale speso dal programma, sia $T_A = \alpha T$ il tempo speso dal programma in A e sia $T_B = (1 - \alpha)T$ il tempo speso in B . Supponiamo di ottimizzare A in modo che sia k volte più veloce. Lo speedup ottenuto è: $S = \frac{1}{\frac{\alpha}{k} + 1 - \alpha}$.

Dimostrazione. Si ha: $T' = \frac{T_A}{k} + T_B = \frac{\alpha T}{k} + (1 - \alpha)T = (\frac{\alpha}{k} + 1 - \alpha)T$, da cui si ottiene: $S = \frac{T}{T'} = \frac{T}{(\frac{\alpha}{k} + 1 - \alpha)T} = \frac{1}{\frac{\alpha}{k} + 1 - \alpha}$.

Esempio.

Di quanto migliorano le prestazioni di un programma se dimezziamo il tempo di esecuzione di una sua porzione che richiede il 40% del tempo totale di esecuzione? Abbiamo $k = 2x$ e $\alpha = 0.4$. Applicando la legge di Amdahl otteniamo: $S = \frac{1}{\frac{0.4}{2} + 1 - 0.4} = \frac{1}{0.8} = 1.25x$.

6.3.2 Profilazione delle prestazioni

La profilazione delle prestazioni consiste nell'analizzare il tempo richiesto dalle singole parti di un programma durante una sua particolare esecuzione. Una prima semplice tecnica per studiare il tempo speso da una determinata porzione di codice è quella di instrumentare il codice stesso inserendo chiamate a system call di misurazione del tempo. Un'alternativa che consente di misurare automaticamente il tempo speso in varie porzioni di codice è mediante l'uso di **profiler**, tool in grado di analizzare l'esecuzione di programmi raccogliendo informazioni utili per studiarne le prestazioni in dettaglio.

6.3.2.1 System call di misurazione del tempo

I sistemi POSIX offrono varie chiamate a sistema per la misurazione del tempo. Alcune, come `clock` e `getrusage` consentono di misurare il tempo speso dal sistema nell'esecuzione sulla CPU di un particolare processo in modalità utente o sistema (ad esempio durante l'esecuzione delle chiamate a sistema). Queste chiamate non permettono tuttavia di contabilizzare il tempo in cui un processo è rimasto in stato waiting. Altre, come `gettimeofday` e `clock_gettime` misurano il tempo reale, anche chiamato wall clock time, calcolando il tempo trascorso rispetto a un determinato istante nel tempo chiamato epoca (UNIX epoch time, o POSIX time), convenzionalmente fissato alle 0:00:00 UTC del 1 gennaio 1970, meno i secondi intercalare (leap seconds) usati come correzione per mantenerlo allineato al giorno solare medio. Altre system call sono orientate dal calcolo delle date e non sono di interesse nel nostro contesto.

Un semplice schema di misurazione del tempo richiesto da una porzione di codice è il seguente, usando una primitiva che chiamiamo convenzionalmente `misura_tempo`:

```
long start = misura_tempo(); // in alternativa, usare double
// porzione di codice da misurare
// ...
long elapsed = misura_tempo() - start;
```

```
printf("Tempo richiesto: %ld\n", elapsed);
```

Vi sono due aspetti rilevanti che caratterizzano una primitiva di misurazione del tempo:

- la **risoluzione**, cioè il minimo intervallo di tempo misurabile
- la **latenza**, cioè il tempo richiesto dall'esecuzione della funzione stessa di misurazione del tempo

E' importante tenere conto di entrambe le caratteristiche nel determinare la **misurabilità di un intervallo di tempo**. In particolare, la durata da misurare dovrebbe essere almeno di un ordine di grandezza maggiore sia della risoluzione che della latenza per ridurre l'errore di misurazione sotto il 10%. Se la **porzione da misurare richiede troppo poco tempo**, si può pensare di **ripeterla** un certo numero di volte, calcolando il tempo totale e dividendolo poi per il numero di ripetizioni per ottenere il tempo medio richiesto dalla porzione. Si tenga presente che ripetere l'esecuzione potrebbe introdurre effetti distorsivi come la presenza di dati in cache caricati dall'esecuzione precedente che potrebbero influenzare le misurazioni.

Vediamo ora come utilizzare la primitiva `clock_gettime`:

```
#include <time.h>
int clock_gettime(clockid_t clk_id, struct timespec *tp)
```

Descrizione: la funzione riporta il tempo in nanosecondi intercorso dalla UNIX epoch

Parametri:

- `clockid_t clk_id`: identificatore del clock da utilizzare per il calcolo. Il valore consigliato da utilizzare per misurare intervalli di tempo in modo affidabile è `CLOCK_MONOTONIC`. Si rimanda alla [documentazione online](#) per altre opzioni.
- `struct timespec tp`: puntatore a un buffer che riceve il tempo misurato;

```
struct timespec
```

Attributi:

- `tv_sec`: secondi misurati
- `tv_nsec`: nanosecondi misurati, da sommare ai secondi

Risultato:

- In caso di successo, restituisce 0
- In caso di errore, restituisce -1 ed `errno` specifica i dettagli dell'errore

Esempio: riportiamo infine un'applicazione della funzione di misurazione, che calcola il tempo richiesto da una funzione di ordinamento. E' possibile provare con numeri diversi di elementi da ordinare per studiare la precisione della misurazione.

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

```

#include <assert.h>
#include <time.h> // clock_gettime

#define NUM 5000 // provare con 5

double get_real_time_msec() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec*1E03 + ts.tv_nsec*1E-06;
}

void sort(int *v, int n) {
    int i, j;
    for (i=0; i<n; ++i)
        for (j=1; j<n; ++j)
            if (v[j-1] > v[j]) {
                int tmp = v[j-1];
                v[j-1] = v[j];
                v[j] = tmp;
            }
}

int main() {
    int i, *v = malloc(NUM*sizeof(int));
    assert(v != NULL);

    for (i=0; i<NUM; ++i) v[i] = NUM-i-1;

    double start = get_real_time_msec();
    sort(v, NUM);
    double elapsed = get_real_time_msec() - start;

    printf("Tempo richiesto da sort: %f msec\n", elapsed);

    free(v);

    return EXIT_SUCCESS;
}

```

6.3.2.2 gprof

`gprof` è uno dei tool di profilazione dei programmi più usati in ambito Linux/UNIX. Consente di **misurare il tempo speso in ciascuna funzione** di un programma in modo da identificare funzioni **hot**, cioè funzioni responsabili per la maggior parte del tempo di esecuzione.

Per utilizzare `gprof` sono richiesti i seguenti passi:

1. Il programma da profilare va **compilato con gcc e l'opzione `-pg`**. Verranno profilate le sole funzioni compilate con quest'opzione.

2. Il **programma** viene **eseguito normalmente**. Durante l'esecuzione, che potrebbe risultare leggermente rallentata, vengono raccolti **profili di esecuzione** che vengono salvati in un file binario `gmon.out`.
3. Una volta terminata l'esecuzione, è possibile **visualizzare il report** generato da `gprof` utilizzando il comando `gprof nome-eseguibile`, dove `nome-eseguibile` è il file eseguito.

Esempio.

Consideriamo il seguente semplice programma:

test-profile.c

```
#include <string.h>
#include <stdlib.h>

#define M 100000
#define N 10000

void init(int *v, int n) {
    memset(v, n*sizeof(int), 255);
}

void A(int *v, int n) {
    int i = 0;
    for (i=0; i<n; i++) v[i] = v[i] / 2;
}

void B(int *v, int n) {
    int i = 0;
    for (i=0; i<n; i++) v[i] = v[i] >> 1;
}

int main() {
    int* v = malloc(N*sizeof(int)), i;
    init(v, N);
    for (i=0; i<M; i++) {
        A(v, N);
        B(v, N);
    }
    free(v);
    return 0;
}
```

Oltre al `main`, il programma contiene tre funzioni che lavorano su un array modificandolo.

Vediamo ora come profilare il programma per studiare quanto tempo viene speso nelle varie funzioni del programma.

Linux: uso di gprof

Compiliamo il programma con l'opzione `-pg`:

```
$ gcc test-profile.c -pg -O1 -o test-profile
```

Per **profilare il programma**, basta eseguirlo:

```
$ ./test-profile
```

A questo punto, dovrebbe essere stato generato un file di report `gmon.out` che contiene le informazioni sul tempo speso in ciascuna funzione:

```
$ ls
gmon.out  test-profile  test-profile.c
```

Per **visualizzare il report** si usa il comando `gprof`:

```
$ gprof test-profile > report.txt
```

Si noti che `> report.txt` serve per redirigere in un file `report.txt` l'output del profiler, che altrimenti verrebbe scritto nella console rendendolo più difficile da consultare. Aprendo il report in editor di testo, notiamo la presenza di una prima tabella che fornisce i tempi spesi in ciascuna funzione. Vi è una seconda tabella più dettagliata che non trattiamo in questa dispensa.

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	us/call	us/call	name
63.86	0.56	0.56	100000	5.56	5.56	A
37.15	0.88	0.32	100000	3.23	3.23	B
0.00	0.88	0.00	1	0.00	0.00	init

Il report contiene istruzioni su come leggere questa tabella. Ci concentriamo in particolare su:

- `% time`: è la percentuale del tempo totale di esecuzione usato dalla funzione;
- `self seconds`: il tempo in secondi speso nella funzione, escludendo il tempo speso nelle funzioni chiamate;
- `calls`: numero di volte che la funzione è stata chiamata;
- `total us/call`: tempo medio per chiamata, incluso il tempo speso nelle sottochiamate, in millisecondi;
- `name`: nome della funzione.

Osserviamo che nella funzione `A` viene speso il 64% del tempo totale mentre in `B` viene speso il 37%. La funzione più onerosa, su cui si concentrerebbero gli sforzi di ottimizzazione, è quindi `A`. Entrambe le funzioni risultano chiamate 100000 volte.

Si noti che `gprof` effettua delle **misurazioni approssimate**, con precisione di 0.01 secondi. Funzioni che richiedono meno di 10 millisecondi non sono pertanto rappresentate. Ne consegue che ha senso usare `gprof` solo se l'esecuzione del programma è sufficientemente lunga da consentire la raccolta di dati significativi.

Bibliografia

[Chaitin et al. 1981] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

Appendice A: tabella dei caratteri ASCII a 7 bit

Le seguenti tabelle contengono i 127 caratteri della codifica [ASCII](#) base (7 bit), divisi in caratteri di controllo (da 0 a 31) e caratteri stampabili (da 32 a 126). Nella tabella riportiamo i codici numerici associati ai caratteri in varie basi (DEC=10, OCT=8, HEX=16).

A.1 Caratteri ASCII di controllo

I primi 32 caratteri sono caratteri non stampabili utilizzati storicamente per controllare periferiche come stampanti. Fra di essi, ci sono i codici che rappresentano i ritorni a capo (caratteri 10 e 13) e il carattere di tabulazione (carattere 9). Nella colonna C riportiamo i codici di escape usati nel linguaggio C per rappresentare alcuni dei caratteri di controllo nelle stringhe e nei letterali `char`.

DEC	OCT	HEX	Simbolo	C	Descrizione
0	000	00	NUL	\0	Null char
1	001	01	SOH		Start of Heading
2	002	02	STX		Start of Text
3	003	03	ETX		End of Text
4	004	04	EOT		End of Transmission
5	005	05	ENQ		Enquiry
6	006	06	ACK		Acknowledgment
7	007	07	BEL	\a	Bell
8	010	08	BS	\b	Back Space
9	011	09	HT	\t	Horizontal Tab
10	012	0A	LF	\n	Line Feed
11	013	0B	VT	\v	Vertical Tab
12	014	0C	FF	\f	Form Feed
13	015	0D	CR	\r	Carriage Return
14	016	0E	SO		Shift Out / X-On
15	017	0F	SI		Shift In / X-Off
16	020	10	DLE		Data Line Escape

17	021	11	DC1		Device Control 1 (oft. XON)
18	022	12	DC2		Device Control 2
19	023	13	DC3		Device Control 3 (oft. XOFF)
20	024	14	DC4		Device Control 4
21	025	15	NAK		Negative Acknowledgement
22	026	16	SYN		Synchronous Idle
23	027	17	ETB		End of Transmit Block
24	030	18	CAN		Cancel
25	031	19	EM		End of Medium
26	032	1A	SUB		Substitute
27	033	1B	ESC	\e	Escape
28	034	1C	FS		File Separator
29	035	1D	GS		Group Separator
30	036	1E	RS		Record Separator
31	037	1F	US		Unit Separator

A.2 Caratteri ASCII stampabili

Vi sono 95 caratteri stampabili, con codici compresi tra 32 e 126:

DEC	OCT	HEX	Simbolo	DEC	OCT	HEX	Simbolo
32	040	20	<i>spazio</i>	80	120	50	P
33	041	21	!	81	121	51	Q
34	042	22	"	82	122	52	R
35	043	23	#	83	123	53	S
36	044	24	\$	84	124	54	T
37	045	25	%	85	125	55	U
38	046	26	&	86	126	56	V

39	047	27	'	87	127	57	W
40	050	28	(88	130	58	X
41	051	29)	89	131	59	Y
42	052	2A	*	90	132	5A	Z
43	053	2B	+	91	133	5B	[
44	054	2C	,	92	134	5C	\
45	055	2D	-	93	135	5D]
46	056	2E	.	94	136	5E	^
47	057	2F	/	95	137	5F	_
48	060	30	0	96	140	60	`
49	061	31	1	97	141	61	a
50	062	32	2	98	142	62	b
51	063	33	3	99	143	63	c
52	064	34	4	100	144	64	d
53	065	35	5	101	145	65	e
54	066	36	6	102	146	66	f
55	067	37	7	103	147	67	g
56	070	38	8	104	150	68	h
57	071	39	9	105	151	69	i
58	072	3A	:	106	152	6A	j
59	073	3B	;	107	153	6B	k
60	074	3C	<	108	154	6C	l
61	075	3D	=	109	155	6D	m
62	076	3E	>	110	156	6E	n
63	077	3F	?	111	157	6F	o
64	100	40	@	112	160	70	p

65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O

113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	
125	175	7D	}
126	176	7E	~

Appendice B: il file system

Il **file system** denota l'insieme di funzionalità e schemi organizzativi con cui un sistema di calcolo si interfaccia con i dispositivi di archiviazione, consentendone la gestione dei dati.

B.1.1 File e directory

Un **file** (archivio) è una sorgente (o un deposito) di informazioni accessibili in lettura e/o in scrittura a un programma. Normalmente è costituito da una sequenza di byte memorizzati in forma permanente su disco.

I file sono organizzati in **directory**, che sono dei contenitori che possono contenere file e altre directory.

File e directory hanno un **nome** che li identifica e altre **proprietà**, fra cui un elenco di **permessi** che specificano quali utenti abbiano diritto di leggerli/scriverli.

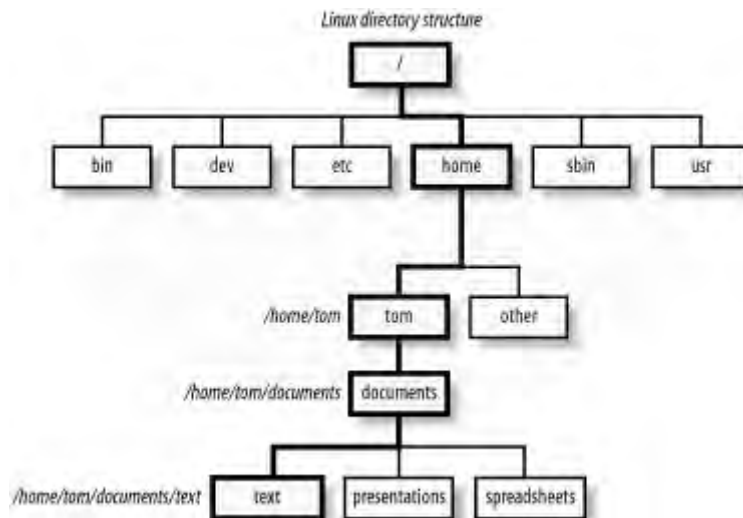
I nomi dei file hanno spesso delle **estensioni** della forma .estensione, che descrivono la natura del loro contenuto. Ad esempio, l'estensione .c denota file di testo scritti in C (es. hello.c), l'estensione .jpg indica file binari che contengono immagini (es. [foto-montagna.jpg](#)), ecc.

B.1.1 L'albero delle directory

La struttura delle directory è tipicamente ad albero: la **directory radice**, denotata da /, contiene tutti i dati memorizzati in forma permanente nel sistema sotto forma di file e sottodirectory.

In questo corso useremo come caso di studio i sistemi [UNIX](#) e [UNIX-like](#) (Linux, MacOS X, ecc.), denotati con l'abbreviazione *nix.

L'organizzazione tipica delle directory di un sistema *nix, chiamata [Filesystem Hierarchy Standard](#) (FHS), è la seguente (semplificata):



Ogni directory, tranne la directory radice ha una **directory genitore**. Se una directory è contenuta in un'altra, allora è una **directory figlia**. Nell'esempio sopra, `tom` e `other` sono figlie di `home`, mentre `home` è genitore di `tom` e `other`. Si noti che i file system UNIX sono tipicamente case-sensitive, pertanto, ad esempio, il file `/home/tom/text` è un oggetto distinto dal file `/home/tom/Text`.

B.1.2 Percorso assoluto e percorso relativo

Ogni file o directory è identificato da un **percorso** (path), che ne identifica la posizione nella struttura delle directory. Un percorso può essere assoluto o relativo.

Percorso assoluto

Elenca tutte le directory che bisogna attraversare per arrivare al punto desiderato a partire dalla directory radice `/`.

Esempio: `/home/anna/Scrivania/foto-montagna.jpg` è il percorso assoluto di un file chiamato `foto-montagna.jpg` posizionato nella directory `Scrivania` localizzata nella `home` directory dell'utente `anna`.

Percorso relativo

Descrive la posizione relativa di una directory rispetto ad un'altra presa come riferimento.

Esempio: il percorso relativo del file `/home/anna/Scrivania/foto-montagna.jpg` rispetto alla directory `/home` è: `anna/Scrivania/foto-montagna.jpg`.

Ci sono due percorsi relativi particolari:

- **.. (doppio punto):** è un percorso relativo che denota la **directory genitore**. Ad esempio, il percorso relativo `../Documenti` può essere letto come: sali alla directory

genitore e poi entra nella directory figlia Documenti. Relativamente a `/home/anna/Scrivania, ../Documenti` denota `/home/anna/Documenti`.

- **.** (**punto**): è un percorso relativo che denota la **directory stessa**. Ad esempio, il percorso `./hello` denota il file chiamato `hello` nella directory di riferimento.

Appendice C: la shell dei comandi

Una **shell** è un programma che consente l'immissione in forma testuale di comandi che devono essere eseguiti dal sistema di calcolo, realizzando quella che viene chiamata **interfaccia a riga di comando** (in inglese: command-line interface, o CLI). In questa dispensa usiamo come shell il programma [bash](#), usato come shell di default in MacOS X e nelle maggiori distribuzioni Linux.

Aprendo una finestra di terminale, si attiva una shell che mostra la **riga di comando**: nella shell `bash` la riga di comando è normalmente indicata dal simbolo `$` seguito dal cursore `|`. Lo scopo del simbolo `$` è quello di avvisare l'utente che la shell è **pronta a ricevere comandi**. Il simbolo `$` è normalmente preceduto da informazioni sull'utente che sta lavorando, sul nome del computer e sulla directory corrente.

Esempio:

```
studente@c1565:~$ |
```

`studente` è il nome dell'utente autenticato, `c1565` è il nome del computer e `~` è la directory corrente (home dell'utente).

Come osservato, in ogni istante la shell è posizionata in una **directory corrente**. All'avvio del terminale, la directory corrente è normalmente la directory home dell'utente con cui ci si è autenticati, indicata dal simbolo `~`. La home directory raccoglie tutti i file, le directory e le impostazioni dell'utente.

Ogni **comando** ha la forma: `nome-comando [parametri]`.

Per far **eseguire un comando** alla shell, lo si digita nel terminale e poi si preme il tasto Invio (Return) ↵. I parametri sono opzionali.

Vi sono due tipi di comandi:

- **Comandi esterni**: `nome-comando` è il nome di un file eseguibile. L'esecuzione del comando lancia un nuovo processo basato sull'eseguibile indicato;
- **Comandi interni (built-in)**: `nome-comando` è un comando eseguito direttamente dalla shell e non provoca l'esecuzione di nuovi processi.

I **percorsi relativi** sono sempre **riferiti** alla **directory corrente** della shell.

Segue un elenco dei comandi interni ed esterni più comunemente usati.

Comando	Tipo	Descrizione
pwd	interno	visualizza il percorso assoluto della directory corrente
cd	interno	cambia directory corrente
ls	esterno	elenca il contenuto di una directory
touch	esterno	crea un file vuoto o ne aggiorna la data di modifica
mv	esterno	rinomina o sposta un file o una directory
mkdir	esterno	crea una nuova directory vuota
rmdir	esterno	elimina una directory, purché sia vuota
rm	esterno	elimina un file o una directory
cp	esterno	copia un file o una directory

C.0 Invocazione di un comando

Come visto, esistono due tipologie di comandi: comandi interni (built-in) e comandi esterni.

All'inserimento da parte dell'utente di un comando, la shell verifica se tale comando risulta essere un percorso relativo o un percorso assoluto ad un binario memorizzato nel file system. In tal caso, tale binario (comando esterno) viene eseguito. Altrimenti, se il comando non riporta un percorso assoluto o relativo ad uno specifico binario, allora la shell esegue una serie di operazioni per identificare quale funzione built-in o binario occorra eseguire. Tali operazioni sono:

1. La shell verifica se il comando matcha il nome di un *alias* definito dal sistema o dall'utente. Un alias è scorciatoia che permette di eseguire dei comandi, anche molto complessi e con diversi parametri, in modo più diretto e conciso. Ad esempio, l'alias "foo=cmd --opzione" permette di eseguire "cmd --opzione" semplicemente digitando "foo". Se un match con un alias viene rilevato, il comando associato all'alias viene eseguito ed i successivi step non vengono eseguiti. Si noti che se il comando specificato dall'alias non contiene un percorso assoluto e relativo, la shell ripete questa procedura ricorsivamente per identificare il binario o built-in associato al comando nell'alias.
2. La shell verifica se il comando matcha il nome di una delle funzioni built-in implementate internamente dalla shell. Se un match con un built-in viene rilevato, esso viene eseguito ed il successivo step non viene eseguito.
3. La shell cerca un binario nel filesystem il cui nome sia equivalente al comando digitato. Tale ricerca viene effettuato solo in alcune specifiche directory del file system. Tali directory sono elencate nella variabile d'ambiente *PATH*. Si noti che una variabile di ambiente può essere vista come una variabile di un programma C e può essere tipicamente manipolata dall'utente, cambiando ad esempio la configurazione della shell.

Tipicamente, in un sistema Linux, la variabile PATH viene inizializzata con la seguente stringa:

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

Tale stringa definisce un insieme di directory, concatenando i loro percorsi utilizzando il delimitatore “:”. Si noti che se il comando ha un nome che matcha con diversi binari (memorizzati all’interno di directory distinte), allora viene selezionato la prima directory su cui si rileva un match. Le directory vengono analizzate nello stesso ordine con cui sono specificate in PATH.

Risulta particolarmente utile il comando built-in `type`, in quanto permette all’utente di verificare quale alias, funzione built-in, o comando esterno si andrà ad eseguire dato un certo comando.

`type`

Indica quale alias, comando built-in, o binario esterno si andrà ad eseguire dato un determinato comando.

Esempio 1:

```
$ type type
type is a shell builtin
```

Il comando `type` è un comando built-in offerto dalla shell.

Esempio 2:

```
$ type ls
ls is aliased to `ls --color=auto'
```

Il comando `ls` è un alias che comporta l’esecuzione del comando `ls --color=auto`. Tale comando verrà analizzato dalla shell, senza considerare il precedente alias (al fine di evitare ricorsioni infinite). In particolare, verrà localizzato il comando `ls` e sarà invocato con l’opzione `--color=auto`.

Esempio 3:

```
$ type cp
cp is /bin/cp
```

Il comando `cp` è un comando esterno che comporta l’esecuzione del binario `/bin/cp`.

Si noti che in ogni momento si può forzare l’esecuzione di uno specifico binario, anche nel caso in cui il binario abbia un nome compatibile con un alias o un comando built-in, semplicemente utilizzando il suo percorso assoluto o un suo percorso relativo valido.

C.1 Manipolazione ed esplorazione del file system

C.1.1 [pwd](#): visualizza il percorso assoluto della directory corrente

```
pwd
```

Visualizza il percorso assoluto della directory corrente.

Esempio:

```
$ pwd  
/home/studente/Scrivania
```

C.1.2 [cd](#): cambia directory corrente

```
cd nome-directory
```

Usa come directory corrente quella specificata dal percorso (assoluto o relativo) `nome-directory`.

Esempio 1:

```
$ cd /home/studente
```

posiziona la shell nella directory `/home/studente`.

Esempio 2:

```
$ cd ..
```

posiziona la shell nella directory genitore di quella corrente.

Esempio 3:

```
$ cd ../Scrivania
```

posiziona la shell nella directory `Scrivania` contenuta nella genitore di quella corrente.

```
cd
```

Posiziona la shell nella home directory dell'utente corrente.

C.1.3 [ls](#): elenca il contenuto di una directory

```
ls [nome-directory]
```

Elenca il contenuto directory specificata dal percorso (assoluto o relativo) `nome-directory`. Se `nome-directory` è assente, elenca il contenuto della directory corrente.

```
ls -l [nome-directory]
```

Elenca il contenuto della directory corrente, fornendo maggiori informazioni (se è un file o directory, la dimensione del file, la data di modifica e altro). Se `nome-directory` è assente, elenca il contenuto della directory corrente.

Esempio output 1:

```
-rw-r--r--    9 anna  staff      306 Oct  8 18:10 hello.c
```

indica (fra altre cose) che `hello.c` è un file e non una directory (la riga inizia per `-`), può essere letto e scritto dall'utente `anna` (`rw`) occupa 306 byte, ed è stato modificato l'8 ottobre alle 18:10. Vedere la sezione sui permessi UNIX per una maggiore discussione di quest'ultimi.

Esempio output 2:

```
drwxr-xr-x    8 studente  staff      272 Sep 27 13:16 foto
```

indica (fra altre cose) che `foto` è una directory (la riga inizia per `d`) può essere letta, scritta e listata dall'utente `studente` (`rw`), e il suo contenuto è stato modificato il 27 settembre alle 13:16.

C.1.4 [touch](#): crea un file vuoto o ne aggiorna la data di modifica

```
touch nome-file
```

Crea un file vuoto `nome-file` o ne aggiorna la data di modifica se esiste già.

Esempio 1:

```
$ touch hello.c
```

crea il file vuoto `hello.c` nella directory corrente o ne aggiorna la data di modifica se esiste già.

Esempio 2:

```
$ touch /home/studente/Scrivania/hello.c
```

crea il file vuoto `hello.c` sulla scrivania dell'utente `studente`, o ne aggiorna la data di modifica se esiste già.

C.1.5 [mv](#): rinomina o sposta un file o una directory

```
mv sorgente destinazione
```

Rinomina il file o la directory sorgente con il nuovo nome destinazione, **purché non esista già una directory con il nome destinazione.**

Esempio 1:

```
$ mv hello.c hello2.c
```

rinomina il file `hello.c` nella directory corrente con il nuovo nome `hello2.c`.

Esempio 2:

```
$ mv pippo pluto
```

rinomina la directory `pippo` con il nuovo nome `pluto`, assumendo che non esista già nella directory corrente una directory chiamata `pluto`.

```
mv sorgente directory
```

Sposta il file o la directory sorgente nella directory directory.

Esempio 1:

```
$ mv hello.c pippo
```

sposta il file `hello.c` nella directory `pippo` (assumendo che `hello.c` e `pippo` siano nella directory corrente).

Esempio 2:

```
$ mv pluto pippo
```

sposta la directory `pluto` nella directory `pippo` (assumendo che `pluto` e `pippo` siano nella directory corrente).

C.1.6 [mkdir](#): crea una nuova directory vuota

```
mkdir directory
```

Crea una nuova directory vuota directory.

Esempio 1:

```
$ mkdir pippo
```

crea la directory `pippo` nella directory corrente.

Esempio 2:

```
$ mkdir /home/studente/Scrivania/pippo
```

crea la directory `pippo` nella directory `/home/studente/Scrivania`.

C.1.7 [rmdir](#): elimina una directory, purché sia vuota

```
rm -r directory
```

Elimina la directory `directory`, purché sia vuota.

Esempio 1:

```
rm -r pippo
```

elimina la directory `pippo` dalla directory corrente.

Esempio 2:

```
rm -r /home/studente/Scrivania/pippo
```

elimina la directory `pippo` dalla directory `/home/studente/Scrivania`.

Nota: per eliminare directory non vuote, si veda il comando `rm`.

C.1.8 [rm](#): elimina un file o una directory

```
rm file
```

Elimina il file `file`.

Esempio 1:

```
$ rm hello.c
```

elimina il file `hello.c` dalla directory corrente.

Esempio 2:

```
$ rm /home/studente/Scrivania/hello.c
```

elimina il file `hello.c` dalla directory `/home/studente/Scrivania`.

```
rm -rf directory
```

Elimina la directory `directory` e tutto il suo contenuto di file e sottodirectory.

Esempio 1:

```
$ rm -rf pippo
```

elimina la directory `pippo` e tutto il suo contenuto dalla directory corrente.

Esempio 2:

```
$ rm -rf /home/studente/Scrivania/pippo
```

elimina le directory `pippo` e tutto il suo contenuto dalla directory `/home/studente/Scrivania`.

C.1.9 `cp`: copia un file o una directory

```
cp file nuovo-file
```

Copia il file `file` creandone uno nuovo chiamato `nuovo-file`.

Esempio 1:

```
$ cp hello.c hello-copia.c
```

copia il file `hello.c` creandone uno nuovo chiamato `hello-copia.c`.

Esempio 2:

```
$ cp /home/studente/Scrivania/hello.c ../hello-copia.c
```

copia il file `hello.c` dalla directory `/home/studente/Scrivania` nella directory genitore di quella corrente con il nome `hello-copia.c`.

```
cp -R directory nuova-directory
```

Copia la directory `directory` e tutto il suo contenuto di file e sottocartelle creandone una nuova dal nome `nuova-directory`.

Esempio 1:

```
$ cp -R pippo pluto
```

copia la directory `pippo` e tutto il suo contenuto di file e sottocartelle creandone una nuova chiamata `pluto`.

Esempio 2:

```
$ cp -R /home/studente/Scrivania/pippo /home/studente/Scrivania/pluto
```

copia la directory `pippo` e tutto il suo contenuto creandone una nuova chiamata `pluto` nella directory `/home/studente/Scrivania`.

```
cp file directory-esistente
```

Copia il file `file` nella directory `directory-esistente`.

Esempio 1:

```
$ cp hello.c pippo
```

copia il file `hello.c` creandone una copia nella directory (esistente) `pippo`.

Esempio 2:

```
$ cp /home/studente/Scrivania/hello.c .
```

copia il file `hello.c` dalla directory `/home/studente/Scrivania` nella directory corrente.

```
cp -R directory directory-esistente
```

Copia la directory `directory` e tutto il suo contenuto di file e sottodirectory nella directory esistente `directory-esistente`.

Esempio 1:

```
$ cp -R pippo pluto
```

copia la directory `pippo` e tutto il suo contenuto di file e sottodirectory nella directory esistente `pluto`.

Esempio 2:

```
$ cp -R /home/studente/Scrivania/pippo .
```

copia la directory `pippo` e tutto il suo contenuto dalla directory `/home/studente/Scrivania` nella directory corrente.

```
cp -R directory/ directory-esistente
```

Copia il **contenuto** della directory `directory` inclusi file e sottodirectory nella directory esistente `directory-esistente`.

Esempio 1:

```
$ cp -R pippo/ pluto
```

copia il contenuto della directory `pippo` inclusi file e sottodirectory nella directory esistente `pluto`.

Esempio 2:

```
$ cp -R /home/studente/Scrivania/pippo/ .
```

copia il contenuto della directory `/home/studente/Scrivania/pippo` inclusi file e sottodirectory nella directory corrente.

C.2 Altri comandi utili

[geany](#)

Lancia l'editor di testo `geany`.

`geany &`: apre l'editor di testo `geany`.

`geany file &`: apre il file di testo `file` nell'editor di testo `geany`.

Esempio:

```
geany hello.c &
```

	<p>apre il file <code>hello.c</code> usando l'editor di testo <code>geany</code>.</p>
cat	<p>Invia il contenuto di un file sul canale di output standard (di default è il terminale).</p> <p><code>cat file:</code> invia il contenuto del file <code>file</code> sul canale di output standard (di default è il terminale).</p>
less	<p>Visualizza nel terminale il contenuto di un file, consentendone di scorrere il contenuto con le frecce.</p> <p><code>less file:</code> visualizza il contenuto del file <code>file</code> nel terminale (premere <code>q</code> per uscire).</p>
man	<p>Visualizza nel terminale la documentazione di un comando esterno, di una libreria, di una system call, e di altri componenti del sistema.</p> <p><code>man [section-id] comando:</code> visualizza nel terminale la documentazione relativa al comando. <code>section-id</code> è un parametro opzionale che indica la categoria di documentazione che si vuole visualizzare. Valore tipicamente utili: 1 (Executable programs or shell commands), 2 (System calls (functions provided by the kernel)), 3 (Library calls (functions within program libraries)). Se <code>section-id</code> viene omissso, <code>man</code> visualizza la documentazione trovata nella prima sezione (in ordine crescente) su cui avviene un match.</p> <p><i>Esempio 1:</i> <code>man cp</code> visualizza nel terminale la documentazione del comando <code>cp</code>.</p> <p><i>Esempio 2:</i> <code>man man</code> visualizza nel terminale la documentazione del comando <code>man</code>.</p> <p><i>Esempio 3:</i> <code>man 3 malloc</code> visualizza nel terminale la documentazione relativa alla funzione <code>malloc</code> della libreria C.</p>
env	<p>Comando built-in che visualizza le variabili d'ambiente definite nella shell corrente.</p> <p><i>Esempio:</i> <code>\$ env</code> <code>SHELL=/bin/bash</code> <code>USER=biar</code> <code>PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin</code> <code>[...]</code></p>

head	<p>Mostra le prime <i>n</i> righe di un file. Default <i>n</i>=10.</p> <p><i>Esempio 1:</i> <code>head file.txt</code> visualizza nel terminale le prime dieci righe di <code>file.txt</code>.</p> <p><i>Esempio 2:</i> <code>head -n 5 file.txt</code> visualizza nel terminale le prime 5 righe di <code>file.txt</code>.</p>
tail	<p>Mostra le ultime <i>n</i> righe di un file. Default <i>n</i>=10.</p> <p><i>Esempio 1:</i> <code>tail file.txt</code> visualizza nel terminale le ultime dieci righe di <code>file.txt</code>.</p> <p><i>Esempio 2:</i> <code>tail -n 5 file.txt</code> visualizza nel terminale le ultime 5 righe di <code>file.txt</code>.</p>
which	<p>Localizza il binario che matcha un comando esterno. Non gestisce correttamente comandi built-in o alias.</p> <p><i>Esempio 1:</i> <code>which cp</code> Viene ritornato il percorso <code>/bin/cp</code>.</p>
whereis	<p>Localizza il binario che matcha un comando esterno. Viene inoltre localizzato qualsiasi altro file utile (come la documentazione) relativo al comando esterno indicato. Non gestisce correttamente comandi built-in o alias.</p> <p><i>Esempio 1:</i> <code>whereis cp</code> Viene ritornato il percorso <code>/bin/cp</code> al binario ed il percorso <code>/usr/share/man/man1/cp.1.gz</code> alla sua documentazione.</p>
grep	<p>Comando avanzato che permette di filtrare il contenuto di un file secondo certe condizioni. Solo le righe del file che matchano le condizioni indicate vengono emesse in output.</p> <p><i>Esempio 1:</i> <code>grep parola file.txt</code> Solo le righe di testo in <code>file.txt</code> che contengono la stringa <code>parola</code> vengono emesse in output.</p> <p><i>Esempio 2:</i></p>

	<pre>grep -v parola file.txt</pre> <p>Solo le righe di testo in <code>file.txt</code> che <i>non</i> contengono la stringa <code>parola</code> vengono emesse in output.</p> <p>Esempio 3:</p> <pre>grep -n parola file.txt</pre> <p>Solo le righe di testo in <code>file.txt</code> che contengono la stringa <code>parola</code> vengono emesse in output. Per ogni riga emessa in output, viene riportato il numero di linea nel file originale.</p>
ln	<p>Questo comando permette di creare dei link nel file system. Esistono due tipologie di link: <i>hard</i> link e <i>soft</i> link. Per questione di semplicità di trattazione, non verrà proposta una discussione degli hard link. Si invita il lettore a consultare le risorse esterne per questo argomento. Un soft link (anche detti link simbolico) offre una scorciatoia verso un determinato elemento del file system. Il comando <code>ln</code> permette di creare link simbolici quando usato con l'opzione <code>-s</code>.</p> <p>Esempio 1:</p> <pre>ln -s /home/tom/Desktop/files/code /home/tom/files-with-code</pre> <p>Viene creato un link simbolico al percorso assoluto <code>/home/tom/files-with-code</code> che punta al percorso assoluto <code>/home/tom/Desktop/files/code</code>.</p> <p>Esempio 2:</p> <pre>ln -s /home/tom/Desktop/files/text.txt text.txt</pre> <p>Viene creato un link simbolico con nome <code>text.txt</code> nella directory corrente che punta al percorso <code>/home/tom/Desktop/files/text.txt</code>. Analogamente si potrebbe anche far riferimento al percorso di destinazione del link con un percorso relativo.</p>

C.3 Permessi UNIX e il comando `chmod`

Come visto negli esempi precedenti, il comando `ls`, se invocato con l'opzione `-l`, mostra diverse informazioni riguardo un elemento del file system. Ad esempio, un possibile output è il seguente:

```
$ ls -l
lrwxrwxrwx 1 biar stud 15 ott 12 17:09 last.log -> output/file.log
drwxrwxr-x 2 biar stud 4096 ott 12 17:09 output
-rwxrwxr-x 1 biar stud 0 ott 4 15:43 test
-rw-rw-r-- 1 biar stud 8696 set 30 12:07 test.c
```

Per ogni elemento figlio della directory corrente vengono mostrate le seguenti informazioni:

1. tipo file (primo carattere della prima colonna):

- a. “-”: file normale
 - b. “l”: link simbolico (vedere comando `ln`)
 - c. “d”: directory
2. Permessi (ultimi 9 caratteri della prima colonna)
 3. Numero di hard link verso il file (seconda colonna)
 4. Utente proprietario del file (terza colonna)
 5. Gruppo proprietario del file (quarta colonna)
 6. Dimensione del file (quinta colonna): si noti che per le directory non viene calcolata ricorsivamente la dimensione dei file contenuti nella directory e tipicamente viene mostrato il valore costante 4096.
 7. Data ultima modifica al file (sesta e settima colonna)
 8. Ora ultima modifica al file (ottava)
 9. Nome del file (nona colonna): se il file è un link simbolico viene mostrato anche il percorso di destinazione del link

Per una discussione dettagliata e completa dell’output di `ls -l` vedere la sua [documentazione](#), sezione “*The Long Format*”.

Come visto, i permessi di un file sono mostrati come una stringa di 9 caratteri (ad esempio `rw-rw-r--x`), che specifica i permessi per tre entità (`user` = utente proprietario, `group` = gruppo proprietario, `other` = altri utenti). Il primo gruppo (**blu**) di tre caratteri indica i permessi per l’utente proprietario del file. Il secondo gruppo (**rosso**) di tre caratteri indica i permessi del gruppo proprietario del file. Il terzo gruppo (**verde**) di tre caratteri indica i permessi per tutti gli utenti che non sono l’utente proprietario e non fanno parte del gruppo proprietario.

Per ogni entità, sono possibili i seguenti permessi:

1. `r`: lettura del contenuto di un file. Se directory, permesso di elencare gli elementi contenuti nella directory (ad esempio con il comando `ls`)
2. `w`: scrittura del contenuto di un file. Se directory, nel caso in cui sia presente anche il permesso di esecuzione, è ammesso rinominare, eliminare o rimuovere elementi contenuti nella directory.
3. `x`: esecuzione del file. Se directory, è possibile accedere alla directory (ad esempio con il comando `cd`)

I permessi sono sempre visualizzati nell’ordine `rw-x` ed un simbolo `-` indica l’assenza del permesso che occupa quella specifica posizione. La rappresentazione esplicita dei tre permessi `rw-x` per le tre entità `ugo` (`user`, `group`, `other`) viene comunemente detta notazione simbolica.

Al contrario, è comune indicare i permessi cumulativi per una entità utilizzando una notazione ottale. Il principale vantaggio di tale rappresentazione è che riesce ad esprimere in modo conciso una combinazione di permessi per un’entità. I permessi `rw-x` possono infatti essere mantenuti utilizzando tre bit $b_2b_1b_0$, dove:

- b_2 se settato ad 1 indica il permesso di lettura (`r`)
- b_1 se settato ad 1 indica il permesso di scrittura (`w`)
- b_0 se settato ad 1 indica il permesso di esecuzione (`x`)

Pertanto, si può esprimere la combinazione dei permessi assegnati ad un'entità, calcolando in numero ottale, il valore ottenuto settando gli opportuni bit b_i . Alcuni esempi (dove $2 \rightarrow_8$ indica il cambio di base da binario ad ottale) sono:

- 000 $2 \rightarrow_8$ 0: nessun permesso
- 001 $2 \rightarrow_8$ 1: permesso di esecuzione
- 010 $2 \rightarrow_8$ 2: permesso di scrittura
- 100 $2 \rightarrow_8$ 4: permesso di lettura
- 101 $2 \rightarrow_8$ 5: permesso di lettura ed esecuzione
- 111 $2 \rightarrow_8$ 7: permesso di lettura, scrittura, ed esecuzione

Si noti come nel caso in cui tutti i permessi siano concessi su di un file per un'entità, la cifra massima che si ottiene è 7. Al contrario, se nessun permesso viene concesso, si ottiene la cifra ottale 0. Questa osservazione motiva l'uso della base ottale (per l'appunto, in questa base le cifre assumono valori compresi tra 0 a 7). Naturalmente, per esprimere i permessi di tutte e tre le entità è necessario utilizzare tre distinte cifre ottali, sempre rispettando l'ordine `ugo`. Ad esempio, `514` indicherà i permessi di lettura e scrittura per l'utente proprietario, il permesso di esecuzione per il gruppo proprietario, ed il permesso di esecuzione per i restanti utenti del sistema.

I permessi su di un file possono essere alterati attraverso il comando [chmod](#). Tale comando ammette sia la notazione simbolica, che, la più diffusa, notazione ottale. Vediamo alcuni esempi dell'uso del comando:

```
chmod entities=symbolic-permissions[, entities-2=symb-perms] file
```

Assegna i permessi `symbolic-permissions` alle entità `entities` per il file `file`.
Opzionalmente, ulteriori entità e relativi permessi possono essere indicati.

Esempio 1:

```
$ chmod ugo=rwx file
```

I permessi di lettura, scrittura ed esecuzione sono concessi per tutte e tre le entità.

Esempio 2:

```
$ chmod u=rwx, go=r file
```

I permessi di lettura, scrittura ed esecuzione sono concessi all'utente proprietario. Il permesso di lettura è concesso al gruppo proprietario ed ai restanti utenti del sistema.

Ulteriori varianti nell'uso del comando `chmod` con la notazione simbolica sono spiegati all'interno della [documentazione ufficiale](#).

```
chmod octal-permissions file
```

Assegna i permessi `octal-permissions` per il file `file`. Il numero ottale deve contenere 3

cifre, una per ogni entità.

Esempio 1:

```
$ chmod 750 file
```

I permessi di lettura, scrittura ed esecuzione sono concessi all'utente proprietario. I permessi di lettura e scrittura sono concessi al gruppo proprietario. Infine, nessun permesso è concesso ai restanti utenti del sistema.

Esempio 2:

```
$ chmod 741 file
```

I permessi di lettura, scrittura ed esecuzione sono concessi all'utente proprietario. Il permesso di lettura è concesso al gruppo proprietario. Infine, permesso di esecuzione è concesso ai restanti utenti del sistema.

Ulteriori varianti nell'uso del comando `chmod` con la notazione ottale sono spiegati all'interno della [documentazione ufficiale](#).

C.4 Redirezione dello stdin, stdout, e stderr di un processo

Di default, quando un processo viene avviato in un sistema UNIX, tre canali di comunicazione (anche detti [standard streams](#)) vengono inizialmente creati. I tre canali sono comunemente denominati:

- standard input (stdin): associato al [file descriptor](#) 0, viene mappato di default all'input generato dalla tastiera. Tale canale viene automaticamente utilizzato da diverse funzioni C: ad esempio `gets()`, `getc()`, e `getchar()` leggono dati provenienti da questo canale.
- standard output (stdout): associato al [file descriptor](#) 1, viene mappato di default sul terminale che ha lanciato il processo. Come conseguenza, qualsiasi output emesso su questo canale (ad esempio attraverso `printf()`), verrà stampato sul terminale.
- standard error (stderr): associato al [file descriptor](#) 2, viene mappato di default sul terminale che ha lanciato il processo. Come conseguenza, qualsiasi output di errore emesso su questo canale (ad esempio attraverso `fprintf(stderr, msg)`), verrà stampato sul terminale. Tale canale è comunemente utilizzato per emettere messaggi di errore o di diagnostica.

Si noti che stdout e stderr sono di default mappati sulla stessa destinazione. Per tale motivo, l'output dei due canali sarà tipicamente alternato, o in alcuni casi addirittura mischiato, sul terminale.

La shell UNIX permette di redirigere questi canali per cambiare la loro destinazione. Vediamo alcuni esempi di redirezione dello stdout e dello stderr.


```
cmd id > file
```

Il canale di output identificato dal file descriptor `id` viene rediretto sul file `file`. Se viene ommesso il valore di `id`, viene utilizzato il valore 1. Il contenuto iniziale del file di destinazione viene cancellato.

Esempio 1:

```
$ ls -l 1> file.log
```

```
$ ls -l > file.log
```

In entrambi i casi, l'output (stdout) del comando `ls` viene rediretto su `file.log`. Il contenuto originale del file di destinazione viene perso.

Esempio 2:

```
$ ls -l 1> out.log 2> error.log
```

L'output (stdout) del comando viene rediretto su `out.log`, mentre lo stderr viene rediretto su `error.log`. Il contenuto originale dei due file di destinazione viene perso.

```
cmd id >> file
```

Il canale di output identificato dal file descriptor `id` viene rediretto sul file `file`. Se viene ommesso il valore di `id`, viene utilizzato il valore 1 (stdout). Il contenuto iniziale del file di destinazione *non* viene cancellato e l'output emesso dal programma viene **appeso in coda** al contenuto originale.

Esempio 1:

```
$ ls -l >> file.log
```

L'output (stdout) del comando `ls` viene rediretto su `file.log`. Il contenuto originale del file di destinazione non viene perso e l'output emesso dal programma vi viene appeso.

Esempio 2:

```
$ ls -l 1>> out.log 2>> error.log
```

L'output (stdout) del comando viene rediretto su `out.log`, mentre lo stderr viene rediretto su `error.log`. Il contenuto originale dei due file di destinazione non viene perso e l'output emesso dal programma vi viene appeso.

```
cmd &> file
```

Entrambi i canali di stdout e stderr identificato vengono rediretti sul file `file`. Il contenuto iniziale del file di destinazione viene cancellato. Utilizzare `>>` per appendere al file.

Esempio:

```
$ ls -l &> file.log
```

Stdout e stderr del comando `ls` vengono rediretto su `file.log`. Il contenuto originale del file

di destinazione viene perso.

La UNIX pipe “|” è un meccanismo della shell che permette di redirigere lo stdout di un processo nello stdin di un altro processo. Tale catena (pipeline) di processamento è tipica della filosofia UNIX, dove ogni utility effettua, o almeno dovrebbe effettuare, esclusivamente un'unica funzionalità di base. Obiettivi complessi dovrebbero poter essere raggiunti utilizzando appunto una pipeline composta da diverse utility.

```
cmd-1 | cmd-2
```

Lo stdout del comando `cmd-1` viene rediretto nello stdin del comando `cmd-2`.

Esempio:

```
$ ls -l | grep str
```

L'output del comando `ls` viene rediretto nello stdin del comando `grep`. Tale pipeline, permette di emettere in output (su terminale) solo le righe di testo emesse da `ls` che contengono la stringa `str`.

C.4 Esecuzione in background e terminazione di un processo

L'esecuzione di un comando nella shell è tipicamente un'operazione bloccante: in altre parole, fintantoché il processo avviato non termina la sua esecuzione, la shell non accetterà più comandi da parte dell'utente. Questo perché il processo viene avviato di default nella modalità foreground ed ogni input inserito dall'utente viene ricevuto nello stdin del processo. Come conseguenza, la shell è obbligata ad attendere la conclusione del processo. Per evitare che ciò accada, e che quindi sia possibile continuare ad interagire con la shell, sono possibili due soluzioni:

1. Il processo può essere inizialmente lanciato in foreground dall'utente. Ad esempio:

```
$ cmd arg1 arg2
```

Tuttavia, successivamente l'utente ha la possibilità di bloccare temporaneamente l'esecuzione del processo lanciato utilizzando la shortcut da tastiera `CTRL+z`. Possibile output del terminale:

```
^Z
[id]+  Stopped                  cmd arg1 arg2
```

Dove `id` è un intero positivo. Dopo questa operazione, l'utente può nuovamente inserire comandi nella shell. Per far continuare l'esecuzione del processo bloccato, l'utente ha nuovamente due possibilità:

- a. riportare in foreground l'esecuzione del processo utilizzando il comando `fg`. Ad esempio, per riportare in foreground il processo con `id` pari ad 1:

```
$ fg 1
```

In tale caso, non sarà nuovamente più possibile interagire con la shell.

- b. Oppure, far continuare l'esecuzione del processo in background. In tale modalità, lo `stdin` del programma non sarà più associato all'input ricevuto da tastiera. Per tale motivo, il processo potrà proseguire la sua esecuzione senza interferire con la shell. Tuttavia, in caso di un'eventuale lettura dallo `stdin` da parte del processo, esso rimarrà in attesa fintantoché l'utente non provvederà a promuovere in foreground nuovamente il processo o fornirà dei dati nello `stdin` del processo in modo alternativo. Un processo bloccato può essere mandato in background attraverso il comando `bg`. Ad esempio, per mandare in background il processo con `id` pari ad 1 si può utilizzare il comando:

```
$ bg 1
```

Si noti che in ogni momento si può risalire all'`id` locale dei processi lanciati dalla shell corrente attraverso il comando `jobs`. Ad esempio:

```
$ jobs
[1]+  Stopped                  cmd-1 arg1
[2]-  Running                  cmd-2 &
```

Tale comando mostra che la shell corrente ha attualmente due processi attivi: il primo nello stato bloccato con `id` pari ad 1, mentre il secondo nello stato in esecuzione con un `id` pari a 2.

2. Altrimenti, l'utente può avviare il processo fin da subito in modalità background. Tale operazione può essere effettuata utilizzando l'operatore "&". Ad esempio, per avviare in background il comando `cmd` occorre inserire il comando:

```
$ cmd &
[id] PID
```

Dove `id` è l'identificativo numerico (locale) del processo nella shell corrente. Mentre `PID` è l'identificativo numerico (globale) del processo nell'intero sistema. Per maggiori informazioni sul `PID`, vedere la sezione sui processi della dispensa. Si noti che ad ogni momento si può risalire al `PID` dei processi attivi avviati nella shell corrente utilizzando il comando `ps`. Ad esempio:

```
$ ps
  PID TTY          TIME CMD
19519 pts/21    00:00:00 bash
25818 pts/21    00:00:02 cmd
28936 pts/21    00:00:00 ps
```

Tale output mostra che nella shell corrente (`pts/21`) sono attivi tre processi: la shell stessa, il comando `ps`, ed il comando `cmd`. Per ogni processo, viene visualizzato il suo `PID` ed il tempo CPU utilizzato.

Per terminare un processo, si hanno almeno due possibilità:

1. *Terminazione di un processo in foreground lanciato dalla shell corrente.* E' possibile provare a terminare un processo utilizzando la shortcut da tastiera `CTRL+C`. Tale operazione genera una *richiesta* (segnale) di terminazione. Al processo viene data la possibilità di effettuare delle operazioni di terminazione (che potrebbero richiedere del tempo considerevole). Il processo ha la possibilità di ignorare tale richiesta.
2. *Terminazione di un processo (in qualsiasi stato e lanciato da qualsiasi shell).* Un processo può essere terminato utilizzando il comando `kill`. Ad esempio, per lanciare una richiesta (segnale) di terminazione, analogamente come fatto con la shortcut `CTRL+C`, si può eseguire il seguente comando:

```
$ kill PID
```

Dove `PID` è l'id globale nel sistema associato al processo. Come nel precedente caso, il processo ha la possibilità di ignorare tale richiesta e non cooperare per la terminazione del processo. In tale, se il processo deve essere realmente terminato, si può forzare la chiusura del processo, utilizzando il comando:

```
$ kill -9 PID
```

Questo comando manda una richiesta (segnale) che non può essere ignorata o gestita dal processo che la riceve. Il processo verrà forzatamente terminato dal sistema operativo.

Appendice D: Debugging di una applicazione C

Esistono numerosi strumenti che possono aiutare un programmatore C ad analizzare l'esecuzione di un programma. Tali strumenti sono tipicamente utilizzati per far emergere le cause di errori e di comportamenti inaspettati. L'attività svolta dal programmatore per analizzare il programma è comunemente detta *debugging* (o *debug*) e gli strumenti utilizzati vengono detti *debugging tools*.

D.1 Compilazione di un programma C con informazioni di debugging

Uno dei requisiti necessari per utilizzare la maggior parte degli strumenti di debugging è l'inclusione delle informazioni di debugging durante la fase di compilazione. Per compilare un programma C con le informazioni di debug, si può utilizzare il compilatore `gcc` con l'opzione `-g`. Ad esempio:

```
$ gcc -g programma.c -o programma
```

Il binario ottenuto potrà essere utilizzato normalmente senza alcuna differenza rispetto al binario generato senza informazioni di debug. Tuttavia, il binario risulterà leggermente più grande in termini di dimensione su disco.

D.2 Valgrind: identificazione di memory leak ed accessi alla memoria non validi

Valgrind è uno strumento molto potente per sistemi UNIX che permette di identificare molti errori comuni dei programmi C. Per eseguire un binario sotto Valgrind, è necessario invocare il binario `valgrind` seguito dal comando che si vuole far analizzare da Valgrind. Ad esempio, possiamo eseguire il binario `programma` sotto Valgrind nel seguente modo:

```
$ valgrind --tool=memcheck ./programma arg1 arg2
```

L'opzione `--tool=memcheck` indica a Valgrind quale dei tool inclusi in questo framework deve essere utilizzato per analizzare il binario. Se l'opzione `tool` viene omessa, il tool `memcheck` viene automaticamente utilizzato. In questa dispensa faremo sempre e solo riferimento all'uso di Valgrind con il tool `memcheck`.

L'output di Valgrind sarà tipicamente strutturato nel seguente modo:

```
==22103== Memcheck, a memory error detector
==22103== Copyright (C) 2002-2015, and GNU GPL'd
==22103== Using Valgrind-3.11.0 and LibVEX;
==22103== Command: ./programma arg1 arg2

// Output del programma
==22103== // Eventuali errori segnalati da Valgrind
// Ulteriore Output del programma

==22103==
==22103== HEAP SUMMARY:
==22103==    in use at exit: 19,628 bytes in 12 blocks
==22103==   total heap usage: 229 allocs, 217 frees, 98,068 bytes
allocated
==22103==
==22103== LEAK SUMMARY:
==22103==    definitely lost: 0 bytes in 0 blocks
==22103==    indirectly lost: 0 bytes in 0 blocks
==22103==    possibly lost: 0 bytes in 0 blocks
==22103==    still reachable: 19,628 bytes in 12 blocks
==22103==         suppressed: 0 bytes in 0 blocks
==22103== Rerun with --leak-check=full to see details of leaked memory
==22103==
==22103== For counts of detected and suppressed errors, rerun with: -v
```

```
==22103== ERROR SUMMARY: 0 errs from 0 contexts (suppressed: 0 from 0)
```

Possiamo identificare le seguenti informazioni:

- Informazioni su Valgrind (**verde**): viene mostrato il tool di Valgrind che è in esecuzione (`memcheck`), la licenza del tool, e la versione di Valgrind installata.
- Il comando eseguito sotto Valgrind (**blu**).
- L'output del programma (**nero**) e gli errori rilevati da Valgrind (**rosso**). L'output del programma viene stampato sul terminale come durante un'esecuzione non tracciata. Eventuali errori rilevati da Valgrind sono segnalati nel terminale. Dato che ogni errore è segnalato non appena esso viene rilevato, i messaggi relativi agli errori emessi da Valgrind potrebbero trovarsi all'interno (primo, dopo, o nel mezzo) dell'output del programma.
- Report sull'uso della memoria heap (**marrone**): Valgrind traccia le invocazioni di `malloc()`, `calloc()`, `realloc()`, e di `free()`. Eventuale memoria allocata ma mai deallocata viene segnalata da `memcheck` nel report finale (`HEAP SUMMARY` e `LEAK SUMMARY`).

Si noti che, al fine di poter discriminare l'output del programma dall'output di Valgrind, ogni linea di testo emessa da Valgrind (e non dal programma originale) viene sempre preceduta dalla stringa `==PID==` (nel nostro esempio: `==22103==`).

Valgrind viene tipicamente utilizzato per rilevare le seguenti categorie di errori:

- *Lettura di un'area di memoria non inizializzata.* Ogni cella di memoria che viene letta da un programma deve essere stata precedentemente inizializzata direttamente o indirettamente dal programma stesso. In caso contrario, il contenuto della cella non è definito ed un qualsiasi valore (non noto a priori) potrebbe essere restituito dalla memoria.

Esempio 1:

unitialized-var.c

```
1 #include <stdio.h>
2
3 int main() {
4     int a;
5     printf("%d\n", a);
6     return 0;
7 }
```

Linux: uso di valgrind

Compiliamo il programma con l'opzione -g:

```
$ gcc uninitialized-var.c -g -o uninitialized-var
```

Eseguiamo il programma:

```
$ ./uninitialized-var  
0x8048461
```

L'output non sembra far emergere alcun problema nel programma. Nonostante ciò, se analizziamo l'esecuzione del programma sotto Valgrind:

```
$ valgrind ./uninitialized-var  
==11028== Memcheck, a memory error detector  
==11028== Copyright (C) 2002-2015, and GNU GPL'd  
==11028== Using Valgrind-3.11.0 and LibVEX;  
==11028== Command: ./uninitialized-ptr  
==11028==  
==11028== Use of uninitialised value of size 4  
==11028==    at 0x40A621B: _itoa_word (_itoa.c:179)  
==11028==    by 0x40AA1F1: vfprintf (vfprintf.c:1631)  
==11028==    by 0x40B05B5: printf (printf.c:33)  
==11028==    by 0x804842B: main (uninitialized-ptr.c:5)  
==11028==  
==11028== Conditional jump or move depends on uninitialised  
value(s)  
==11028==    at 0x40A6223: _itoa_word (_itoa.c:179)  
==11028==    by 0x40AA1F1: vfprintf (vfprintf.c:1631)  
==11028==    by 0x40B05B5: printf (printf.c:33)  
==11028==    by 0x804842B: main (uninitialized-ptr.c:5)  
==11028==  
==11028== Conditional jump or move depends on uninitialised  
value(s)  
==11028==    at 0x40A9774: vfprintf (vfprintf.c:1631)
```

```

==11028==      by 0x40B05B5: printf (printf.c:33)
==11028==      by 0x804842B: main (unitialized-ptr.c:5)
==11028==
==11028== Conditional jump or move depends on uninitialised
value(s)
==11028==      at 0x40A9C8D: vfprintf (vfprintf.c:1631)
==11028==      by 0x40B05B5: printf (printf.c:33)
==11028==      by 0x804842B: main (unitialized-ptr.c:5)
==11028==
==11028== Conditional jump or move depends on uninitialised
value(s)
==11028==      at 0x40A9AD4: vfprintf (vfprintf.c:1631)
==11028==      by 0x40B05B5: printf (printf.c:33)
==11028==      by 0x804842B: main (unitialized-ptr.c:5)
==11028==
==11028== Conditional jump or move depends on uninitialised
value(s)
==11028==      at 0x40A9B1D: vfprintf (vfprintf.c:1631)
==11028==      by 0x40B05B5: printf (printf.c:33)
==11028==      by 0x804842B: main (unitialized-ptr.c:5)
==11028==
134513761
==11028==
==11028== HEAP SUMMARY:
==11028==      in use at exit: 0 bytes in 0 blocks
==11028==    total heap usage: 1 allocs, 1 frees, 1,024 bytes
allocated
==11028==
==11028== All heap blocks were freed -- no leaks are possible
==11028==
==11028== Use --track-origins=yes to see where uninitialised
values come from
==11028== ERROR SUMMARY: 22 errors from 6 contexts

```

Possiamo vedere come Valgrind segnali diversi errori (**testo in rosso**). L'approccio che occorre seguire è quello di analizzare il primo errore, identificarne la causa, correggerlo e ripetere l'esecuzione sotto Valgrind. Infatti, è abbastanza comune che un unico errore nel codice programma possa riflettersi in diverse errori a tempo di esecuzione. Per tale motivo, è consigliabile risolvere un errore alla volta e rivalutare l'output di Valgrind dopo ogni cambiamento del codice sorgente. Il primo errore segnalato riguardo un uso (accesso in lettura), alla riga 5 del codice sorgente, di una variabile di dimensione 4 byte non inizializzata. Valgrind suggerisce (**testo in blu**) all'utente di ripetere l'esecuzione utilizzando l'opzione `--track-origins=yes` per ottenere maggiori informazioni sulla causa degli errori:

```

$ valgrind --track-origins=yes ./unitialized-var
[...]
==11105== Use of uninitialised value of size 4
==11105==      at 0x40A621B: _itoa_word (_itoa.c:179)

```



```

==11105==      by 0x40AA1F1: vfprintf (vfprintf.c:1631)
==11105==      by 0x40B05B5: printf (printf.c:33)
==11105==      by 0x804842B: main (unitialized-ptr.c:5)
==11105== Uninitialised value was created by a
              stack allocation
==11105==      at 0x804841C: main (unitialized-ptr.c:5)
[...]
```

Con questa opzione, Valgrind è in grado di risalire all'origine del problema: una variabile locale (`stack allocation`), creata alla linea 5 (in realtà alla riga 4, purtroppo Valgrind non sempre riesce ad essere preciso), non è mai stata inizializzata ma viene letta dal programma, generando un accesso in lettura non valido. Una possibile modifica risolutiva del problema consiste nel modificare la riga 4 nel seguente modo:

```
4    int a = 1;
```

Eseguendo il programma modificato sotto Valgrind, nessun errore viene più segnalato. Si noti che il valore stampato in output dal programma prima della modifica al sorgente risulta essere un valore spazzatura (*garbage*), ossia un valore memorizzato nella memoria che non può essere predeterminato a priori perché dipende dalla specifico contesto di esecuzione (che varia da macchina a macchina, da piattaforma a piattaforma, da sistema operativo a sistema operativo).

- *Salto condizionato (`if`, `test condizione while`, `test condizione for`) su dati non inizializzati.*

unitialized-conditional-jump.c

```

1 #include <stdio.h>
2
3 void foo(int *a) {
4     if (a == NULL)
5         return;
6     printf("a is %d\n", *a);
7 }
8
9 int main() {
10     int *a;
11     foo(a);
12     return 0;
13 }
```

Linux: uso di valgrind

Eseguiamo il programma:

```
$ ./uninitialized-conditional-jump  
a is -16219251
```

L'output non sembra far emergere alcun problema nel programma. Nonostante ciò, se analizziamo l'esecuzione del programma sotto Valgrind:

```
$ valgrind ./uninit-var  
==24424== Memcheck, a memory error detector  
==24424== Copyright (C) 2002-2013, and GNU GPL'd, by Julian  
Seward et al.  
==24424== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h  
for copyright info  
==24424== Command: ./uninit-var  
==24424==  
==24424== Conditional jump or move depends on uninitialised  
value(s)  
==24424==      at 0x40053E: foo (uninit-var.c:5)  
==24424==      by 0x40056E: main (uninit-var.c:12)  
==24424==  
==24424==  
==24424== HEAP SUMMARY:  
==24424==      in use at exit: 0 bytes in 0 blocks  
==24424==    total heap usage: 0 allocs, 0 frees, 0 bytes  
allocated  
==24424==  
==24424== All heap blocks were freed -- no leaks are possible  
==24424==  
==24424== For counts of detected and suppressed errors, rerun  
with: -v
```

```
==24424== Use --track-origins=yes to see where uninitialised  
values come from  
==24424== ERROR SUMMARY: 1 errors from 1 contexts (suppressed:  
0 from 0)
```

Altri tipi di errori identificabili da Valgrind includono:

- *Accesso (scrittura, lettura, ed esecuzione) di un'area di memoria non valida.*
- *Memory leak (memoria non deallocata).*
- *Invalid free (deallocazione di puntatori non validi).*