**Sistemi di Calcolo**

Ultimo aggiornamento: 4 maggio 2021



**Camil Demetrescu**

**Appendici: 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?**](#_sow1tvdlsfkx) **9**  [1.1 Hardware](#_6cy60i5r3re6) 10  [1.1.1 Panoramica elementi hardware](#_jczrkiconqib) 10  [1.1.1.1 CPU](#_8yqfm4ljp9b0) 10  [1.1.1.2 Bus](#_74hcxqvzz263) 11  [1.1.1.3 I/O bridge](#_rhjw543vyopw) 11  [1.1.1.4 Controller e adattatori](#_a4pa042ymfns) 11  [1.1.1.5 Sistema di memoria](#_sbp4d23udlqm) 11  [1.1.2 Elementi di base di un microprocessore](#_xkozf0qbt8jk) 12  [1.1.2.1 Porte logiche](#_4wd77m5o2vi3) 12  [1.1.2.2 Miniaturizzazione e VLSI](#_o2496eixjptm) 16  [1.1.2.3 Circuiti combinatori](#_w6gi3ugji3h) 18  [1.1.2.4 Logica sequenziale e flip-flop](#_yan51su24pp5) 20  [1.2 Sistema operativo](#_bug4ub95g37y) 22  [1.2.1 Compiti](#_suacswnc02ao) 22  [1.2.2 Obiettivi di progettazione](#_b16ctc6w9f2r) 23  [1.2.3 Panoramica del funzionamento](#_f1aptr1fr3nh) 23  [1.3 Domande riepilogative](#_gp9mdu5lngdd) 26  [**2 Come viene programmato un sistema di calcolo?**](#_m2y8thd9ae3p) **28**  [2.1 Linguaggi di programmazione di alto livello e di basso livello](#_49c94t7li0eb) 28  [2.2 Compilazione vs. interpretazione](#_ipphdwtee9gl) 28  [2.3 Stadi della compilazione di un programma C](#_ch1xmk9lc5ra) 29  [2.3.1. Il toolchain di compilazione gcc](#_4e8spbqfzar1) 31  [2.3.2 Programmi in formato testo, oggetto, ed eseguibile (ELF, Mach-O)](#_185fiio6t0ky) 31  [2.3.3 Disassemblare programmi in codice macchina: objdump](#_eb57tpl9cbz5) 33  [2.4 Domande riepilogative](#_uwkd68kqbwa4) 35  [**3 Come viene tradotto in linguaggio macchina un programma C?**](#_324es8uns83q) **36**  [3.1 Instruction set architecture (ISA) IA32 - Sintassi AT&T](#_x0s0mcjp7tai) 36  [3.1.1 Tipi di dato macchina](#_3pb5pkn40ag6) 36  [3.1.2 Corrispondenza tra tipi di dato C e tipi di dato macchina](#_9hn3gsljudex) 37  [3.1.3 Registri](#_cchndmv1zony) 38  [3.1.4 Operandi e modi di indirizzamento della memoria](#_o842j2osd7e7) 39  [3.1.5 Rappresentazione dei numeri in memoria: big-endian vs. little-endian](#_px6kylw0t3ji) 40  [3.1.6 Istruzioni di movimento dati](#_w14oo7gmr4u) 41  [3.1.6.1 Stessa dimensione sorgente e destinazione: MOV](#_7gs6batjlbu) 41  [3.1.6.2 Dimensione destinazione maggiore di quella sorgente: MOVZ, MOVS](#_3kofrbgq7gfk) 41  [3.1.6.3 Movimento dati da/verso la stack: PUSH, POP](#_toc6evc573z4) 43  [3.1.7 Istruzioni aritmetico-logiche](#_qu16lxp0dhsj) 43  [3.1.7.1 L’istruzione LEA (load effective address)](#_p03uf9vyie09) 45  [3.1.7.2 Istruzioni di shift SHL, SHR, SAL, SAR](#_pwnanxsjwixi) 46  [3.1.8 Istruzioni di salto](#_wx3kxbza8f74) 48  [3.1.8.1 Salti incondizionati: JMP](#_npbk34mw3vht) 48  [3.1.8.2 Salti condizionati e condition code: Jcc, CMP](#_vqvtkhm3ncbn) 49  [3.1.8.3 Chiamata e ritorno da funzione: CALL e RET](#_907f5jv5qsk3) 52  [3.1.9 Altre istruzioni e vincoli sulle istruzioni](#_lv1gszgt2gpg) 54  [3.1.9.1 Istruzioni di assegnamento condizionato: CMOVcc](#_ydcs1786jjac) 54  [3.1.9.2 Altre istruzioni di confronto: TEST](#_ll86u38w22fv) 55  [3.1.9.3 Altre istruzioni di assegnamento: SETcc](#_5e6kcnttvanu) 55  [3.1.9.4 Vincoli sulle istruzioni](#_f0jxzzg2mpw0) 56  [3.2 Traduzione dei costrutti C in assembly IA32](#_t4xep6zc6wmz) 56  [3.2.1 Istruzioni condizionali](#_wv9mh0thg1jg) 56  [3.2.1.1 Istruzione if](#_sg2ya06bj345) 57  [3.2.1.1 Istruzione if…else](#_susddl8xko6n) 57  [3.2.2 Cicli](#_mrmr0a43ez7u) 58  [3.2.2.1 Istruzione while](#_l8xxlho19134) 58  [3.2.2.1 Istruzione for](#_f5d1yncuuhn4) 59  [3.2.3 Funzioni](#_ol3xgnnfosls) 61  [3.2.3.1 Restituzione valore](#_diyklumxa2lu) 62  [3.2.3.2 Stack frame e registro EBP](#_nrl766r7mdgm) 62  [3.2.3.3 Passaggio dei parametri](#_wk51191zg5ci) 64  [3.2.3.4 Registri caller-save e callee-save](#_giatdlobxgn8) 66  [3.2.3.5 Variabili locali](#_34yhx0ggvhmf) 67  [3.2.4 Array e aritmetica dei puntatori](#_u5qgms3a3yuj) 69  [3.2.5 Strutture](#_5kt4a0z1i6gr) 71  [3.3 Estensione a x86-64](#_uzsk4ytkbvs6) 72  [3.3.1 Registri](#_o3mdp3h6e7sp) 72  [3.3.2 Passaggio dei parametri alle funzioni](#_w9hfl64hfuld) 73  [3.3.3 Registri caller-save e callee-save](#_i4emmf6kekwm) 74  [3.3.4 Suffissi delle istruzioni a 64 bit e istruzioni MOVS e MOVZ](#_a5fobd3xay) 74  [3.4 Domande riepilogative](#_j4b0i21xy4v7) 74  [**4 Su quali librerie standard può contare un programmatore C?**](#_asv35a9u9khi) **76**  [4.1 Accesso alla documentazione: man](#_nhwbdk7w67ez) 76  [4.2 Gestione degli errori](#_wegnmd3qafo0) 77  [4.2.1 Codice di terminazione di un programma](#_sd1n9bx4eq7i) 77  [4.2.2 Installare un gestore per la terminazione di un programma](#_cfe8qqk7ogyh) 77  [4.2.3 Errori non recuperabili](#_p0mgbxg7c92h) 78  [4.2.4 Errori recuperabili](#_rl2sarxokgr6) 78  [4.3 Panoramica della libreria standard C](#_7t1zq8sry9hz) 79  [4.3.1 Stringhe: strlen, strcpy, strcat, strcmp, sprintf, sscanf, strtok, atoi](#_yp2u4pry90c5) 79  [4.3.2 Memoria: memcpy, memcmp, memset](#_mqjecm1i15f8) 83  [4.3.3 File e stream](#_er65d0utpnmh) 84  [4.3.3.1 Canali standard di input/output: stdin, stdout, stderr](#_go7qxqbxhz02) 85  [4.3.3.2 Apertura e chiusura di file: fopen e fclose](#_h24qbl9sagvu) 85  [4.3.3.3 Lettura e scrittura di file di testo: printf, fprintf, scanf, fscanf, fgets](#_c7lof5iy3ukf) 86  [4.3.3.4 Lettura e scrittura di file binari: fwrite e fread](#_8qwezpklicwi) 89  [4.3.4 Ricerca e ordinamento: bsearch, qsort](#_hmktm2prlcp0) 90  [4.3.5 Funzioni matematiche](#_yntuy6caff48) 92  [4.4 Tracciamento delle system call: strace](#_6iuex2t5zlpa) 92  [4.5 Tracciamento delle funzioni di libreria: ltrace](#_izd2ersb1pqj) 93  [**5 Come vengono ottimizzati i programmi?**](#_90ym3e6wrugv) **95**  [5.1 Quanto è importante ottimizzare le prestazioni?](#_rrruvg9mzwun) 96  [5.2 Quali parti di un programma ottimizzare e cosa ottimizzare?](#_kgtbxw1ysiy4) 97  [5.2.1 Metriche di prestazioni](#_wz7cqbnn13vb) 97  [5.2.2 Scala degli eventi in un sistema di calcolo](#_d81gpu30ud6b) 98  [5.2.3 Speedup e legge di Amdahl](#_9m8udd9u9sif) 98  [5.2.4 Profilazione delle prestazioni](#_e9ivtl9ikxqk) 99  [5.2.4.1 System call di misurazione del tempo](#_854gnub6i6p7) 100  [5.2.4.2 gprof](#_nj7qziyv5hdx) 103  [5.3 Tecniche di ottimizzazione delle prestazioni di un programma](#_hjfxw6vof8fb) 106  [5.3.1 Livelli di ottimizzazione in gcc](#_hbpg2phe9iqh) 106  [5.3.2 Ridurre il numero di istruzioni eseguite](#_wc1vjpvf2o8z) 107  [5.3.2.1 Constant folding](#_hkospmx6xut7) 107  [5.3.2.2 Constant propagation](#_sqjh9xxoefa) 107  [5.3.2.3 Common subexpression elimination](#_alut3f39vgzq) 108  [5.3.2.5 Loop-invariant code motion (hoisting)](#_1o8wtqmfu75f) 109  [5.3.2.6 Loop unrolling](#_sm4n8jg3cz68) 111  [5.3.2.7 Function inlining](#_j72igd1fbnkz) 112  [5.3.3 Ridurre il costo delle istruzioni eseguite](#_m72tsy453ux5) 113  [5.3.3.1 Register allocation](#_a7fjouqjwkwh) 113  [5.3.3.2 Strength reduction](#_h5bgdi744zx) 114  [5.3.3.3 Allineamento dei dati in memoria](#_1fvnl910x3g7) 115  [5.3.3.4 Altre ottimizzazioni](#_3cxy4pg9hw6v) 116  [5.3.4 Ridurre lo spazio di memoria](#_4h5r1krrm4xm) 116  [5.3.4.1 Ottimizzare lo spazio richiesto dalle strutture C](#_erf384bkpvbl) 117  [5.3.4.2 Dead code elimination](#_qbpdpy4u1n37) 117  [5.3.5 Le ottimizzazioni dei compilatori sono le migliori possibili?](#_2n16c3dv8tsk) 118  [**6 Come vengono eseguiti i programmi?**](#_73zk7k2dfmmy) **119**  [6.1 Processi](#_mn8ek7tzv9qb) 119  [6.1.1 Immagine di memoria di un processo](#_nfatk2cdcocv) 120  [6.2 Esecuzione dei programmi](#_qjzy4glm98ov) 124  [6.2.1 Esecuzione di una singola istruzione](#_ndequ9niow9a) 124  [6.2.2 Esecuzione simultanea di più istruzioni: pipelining](#_nlrvnh7cp79l) 125  [6.2.3 Esecuzione simultanea di più processi](#_mmjeq4sydgah) 128  [6.2.3.1 Stati di un processo, schedulazione, preemption, time-sharing](#_go7u061hqmz2) 128  [6.2.3.2 Creazione di processi: fork](#_nizhp3xydjea) 130  [6.2.3.3 Recupero del pid dei processi: getpid, getppid](#_vx3qs8yrnj6u) 131  [6.2.3.4 Attesa della terminazione di un figlio: wait](#_vr76138jrlhn) 132  [6.2.3.5 Passaggio dei parametri ai programmi: argc e argv](#_um70ndbb8cud) 133  [6.2.3.6 Caricamento di programmi (loader): exec](#_lksdfcgjqvms) 134  [6.3 Flusso del controllo eccezionale](#_s1rpgimmhiy4) 135  [6.3.1 Interruzioni](#_d91f6q7oip4d) 136  [6.3.1.1 Preemption e context switch tra processi](#_qv7c3qe37ox8) 137  [6.3.1.2 System call e loro invocazione con int $0x80](#_nb1ikvyi7bzf) 138  [6.3.1.3 Passaggio di stato di un processo da waiting a ready: interrupt vs. polling](#_iop5ju6kpb8z) 139  [6.3.2 Segnali](#_bct2rew9e7wr) 139  [6.3.2.1 Definizione segnali](#_xn4bozf84qv2) 140  [6.3.1.2 Invio segnali](#_j0ll6mf8yfj4) 141  [6.3.1.4 Gestione segnali: sigaction](#_98fmdqhw9hn5) 142  [6.3.1.3 Attesa segnali: pause](#_53hreh8s1t7p) 145  [6.3.1.4 Timer](#_qm71exqpscgt) 145  [**7 Come vengono gestiti i dati in memoria centrale?**](#_vxh8382ukz33) **147**  [7.1 Come viene allocata la memoria?](#_pwz4o2sckfr8) 147  [7.1.2 Unità di misura della quantità di memoria](#_5mzunt3gpm76) 147  [7.1.3 Allocazione dinamica della memoria](#_r5yv7hx4nn47) 148  [7.1.3.1 Frammentazione interna ed esterna](#_jm6lz0vk8mz4) 149  [7.1.3.2 Qualità di un allocatore: tempo e spazio](#_85z83psb0g6k) 149  [7.1.3.3 Allocazione in cascata della memoria](#_uy9r5q2gdtsy) 149  [7.1.4 Memoria fisica e memoria virtuale](#_o3pkeofg3xnb) 150  [7.1.5 Allocazione nella memoria fisica: memoria virtuale](#_fo77pab1svi2) 150  [7.1.5.1 Mapping tra indirizzi virtuali e indirizzi fisici: MMU](#_4zgf3d4ub0l0) 151  [7.1.5.2 Paginazione](#_fvxq46tfzjj4) 151  [7.1.5.3 Come avviene il mapping degli indirizzi logici su quelli fisici?](#_q33i5pnu5zor) 152  [7.1.5.4 Paginazione con bit di validità](#_5x6mkhs88fkt) 153  [7.1.5.5 Paginazione su richiesta](#_ntwy8b8ziyon) 155  [7.1.6 Allocazione nella memoria logica: malloc e free](#_34kfpvyvp4zh) 155  [7.2 Come ottenere le migliori prestazioni usando la memoria?](#_hltzzpqnkknf) 157  [7.2.1 Memorie cache](#_nhh5lys285s3) 158  [7.2.2 Politiche di rimpiazzo delle linee](#_6wsyp36yjmef) 159  [7.2.3 Località spaziale e temporale](#_ny3fxukre1gr) 159  [7.2.4 Esempi](#_et53m1dt8590) 159  [7.2.4.1 Somma di matrice](#_xkweu27zi6lk) 160  [7.2.4.2 Analisi di una sequenza di accessi](#_ichmb1k23z95) 161  [7.2.5 Associatività](#_qe5e4xamqyag) 161  [7.2.6 Tipi di cache miss](#_osu4uj608lzh) 162  [7.2.7 Gerarchie dTi memoria](#_d08tvc72r2wk) 162  [**8 Come vengono gestiti i file?**](#_m3ql90e45mrf) **164**  [8.1.1 L’albero delle directory](#_69zcnizftauo) 165  [8.1.2 Percorso assoluto e percorso relativo](#_mziy6bvl6bla) 165  [8.2 Organizzazione di un file system](#_qqz845orzeuz) 166  [8.2.1 Inode e blocchi](#_n0bs2vlw4kgi) 166  [8.2.2 Tabella dei descrittori di file di un processo](#_wy1g4supgmsb) 169  [8.3 Mount e unmount di file system sull'albero delle directory](#_llmtfqa8op1) 170  [8.4 Operazioni su file e directory](#_9pgr1a2zbgkl) 171  [8.4.1 Apertura e chiusura: open e close](#_v16m239o28g1) 172  [8.4.2 Scrittura e lettura: write e read](#_ijw4emkim3ht) 173  [8.4.3 Accesso casuale: lseek](#_jqbinr9gjkl7) 174  [8.4.4 Lavorare con le directory](#_ejcsbs32x64g) 175  [8.5 Link](#_8lcm67cnhx8o) 176  [8.5.1 Symbolic link](#_mgg5qxafrnap) 177  [8.5.2 Hard link](#_txda06f2b6i1) 177  [8.5.2.1 Creazione di link: link](#_b9wec7j87eiu) 177  [8.5.2.2 Eliminazione di link: unlink](#_aavmpioq503w) 179  [**Bibliografia**](#_h185c8dlbtfx) **180**  [**Appendice A: tabella dei caratteri ASCII a 7 bit**](#_a85g1vvkqj2z) **181**  [A.1 Caratteri ASCII di controllo](#_hdevmam3ah4p) 181  [A.2 Caratteri ASCII stampabili](#_gomg1z3f4t25) 182  [**Appendice B: la shell dei comandi**](#_w6lesegis88) **185**  [B.1 Invocazione di un comando](#_ygpsi252zy2t) 186  [B.2 Manipolazione ed esplorazione del file system](#_mu1bgxvw43m6) 188  [B.2.1 pwd: visualizza il percorso assoluto della directory corrente](#_304s4hj1jhh1) 188  [B.2.2 cd: cambia directory corrente](#_pb1wxjuzqr0h) 188  [B.2.3 ls: elenca il contenuto di una directory](#_jofw3djc1db1) 189  [B.2.4 touch: crea un file vuoto o ne aggiorna la data di modifica](#_110d7ncn2ehk) 189  [B.2.5 mv: rinomina o sposta un file o una directory](#_p5kj7u8sdp9w) 190  [B.2.6 mkdir: crea una nuova directory vuota](#_yioalpwo29xj) 191  [B.2.7 rmdir: elimina una directory, purché sia vuota](#_4t1fcpx7v1o8) 191  [B.2.8 rm: elimina un file o una directory](#_nbbbywdqvk4e) 191  [B.2.9 cp: copia un file o una directory](#_vxmril5u4qh8) 192  [B.3 Altri comandi utili](#_zie2jyhur84x) 194  [B.4 Permessi Posix e il comando chmod](#_ye1ju02ejb2d) 197  [B.5 Redirezione dello stdin, stdout, e stderr di un processo](#_8fyy8diz8l9k) 200  [B.6 Esecuzione in background e terminazione di un processo](#_ergb5vt1pzju) 201  [**Appendice C: debugging di un programma C**](#_3neuq4l8souv) **204**  [C.1 Compilazione di un programma C con informazioni di debugging](#_6bqu4oom23ss) 204  [C.2 Il debugger gdb](#_j9igcoq9hk37) 204  [C.2.1 Preliminari](#_179fs7vjvd4g) 204  [C.2.2 Compilare programma con i simboli di debugging](#_h8dhxkdudj3h) 205  [C.2.3 Avviare il debugger](#_lyqlocnb08b2) 205  [C.2.4 Comandi](#_sps3en8l371m) 205  [C.2.4.1 Avvio e terminazione](#_10f6lhrw8pfr) 205  [C.2.4.2 Controllo esecuzione](#_qtf6q07tqmyi) 205  [C.2.4.3 Gestione breakpoint](#_ev60kvu3wy9w) 205  [C.2.4.4 Ispezione dello stato](#_gdisglpd5ezm) 206  [C.2.4.5 Formato nfu](#_gcrb97wxy64) 206  [C.2.4.6 Abbreviazioni dei comandi:](#_45bvy6w2jjsk) 206  [C.2.4.7 Altri comandi](#_gf7wqfw92bgu) 206  [C.3 Valgrind: debugging dell'uso della memoria](#_m5pbjf5d9ql) 207  [**Appendice D: prerequisiti di elettronica**](#_wxp7annpbp2w) **213**  [D.1 Resistenze e prima legge di Ohm](#_gdmgholxn4n4) 213  [D.2 Transistor](#_cnb1dvd6m70) 215  [D.3 Breadboard](#_olzt3lmkq1ne) 215 |
| --- |

# 

# 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, Xbox, ecc.), e molto altro ancora che non sospetteremmo possa essere pensato come un sistema di calcolo (Figura 1.1).



**Figura 1.1**: esempi di sistemi 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**.

## 1.1 Hardware

In questo corso tratteremo sistemi di calcolo in cui l’hardware dei singoli nodi è organizzato secondo il **modello di Von Neumann** (Figura 1.1.1).



Figura 1.1.1: macchina 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 dell'hardware di un calcolatore moderno è illustrata in Figura 1.1.2.



**Figura 1.1.2**: macchina di Von Neumann, più in dettaglio

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

### 1.1.1 Panoramica elementi hardware

#### 1.1.1.1 CPU

La **CPU** (o **microprocessore**) è il cuore del sistema. Esso 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 diverse[[1]](#footnote-0). Le istruzioni vengono eseguite dalla CPU nell’ordine in cui appaiono in memoria ed effettuano tipicamente operazioni che:

1. **trasferiscono dati** all’interno della CPU, all’interno della memoria, fra memoria e CPU, e fra dispositivi esterni e la CPU.
2. calcolano operatori **aritmetico/logici** (somme, prodotti, ecc.), operatori **booleani** (congiunzione, disgiunzione, negazione, ecc.), operatori **relazionali** (uguale, maggiore, minore, ecc.). Questi operatori vengono eseguiti da un modulo apposito della CPU noto come Unità Aritmetico-logica (ALU).
3. 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).

#### 1.1.1.2 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.

#### 1.1.1.3 I/O bridge

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

#### 1.1.1.4 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).

#### 1.1.1.5 Sistema di memoria

Il sistema di memoria di un calcolatore tipico (desktop, laptop, server) è molto articolato come illustrato in Figura 1.1.1.1. La CPU non comunica direttamente con la memoria principale (RAM), ma ci sono una serie di strati di memorie intermedie chiamate cache che memorizzano i dati più frequentemente acceduti o adiacenti. Le cache hanno tempi di accesso molto inferiori a quelli di una RAM.

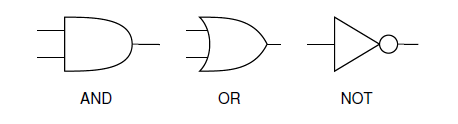


**Figura 1.1.1.1**: sistema di memoria gerarchica di un sistema di calcolo

### 1.1.2 Elementi di base di un microprocessore

#### 1.1.2.1 Porte logiche

Le porte logiche realizzano funzionalità elementari di una CPU. Esse sono elementi di calcolo che realizzano funzioni booleane che a uno o più valori binari di input assegnano un valore binario di output. Le tre porte logiche principali sono AND, OR, NOT e sono rappresentate mediante i simboli mostrati in Figura 1.1.2.1.

****

**Figura 1.1.2.1**: simboli convenzionali corrispondenti alle porte logiche AND, OR e NOT

Il loro comportamento può essere descritto da tabelle di verità come vedremo nei paragrafi seguenti. Inoltre, essi sono normalmente realizzati mediante componenti elettronici basati su semiconduttori come i transistor. Un transistor ha tre connettori: collettore, base ed emettitore. Funziona come un interruttore pilotato dall'elettricità: se la base è raggiunta da corrente, si crea un collegamento elettrico tra collettore ed emettitore. Nei paragrafi seguenti assumeremo che una tensione di 5 Volt corrisponde al valore di verità 1 (true), mentre una tensione elettrica di 0 Volt corrisponde al valore di verità 0 (false).

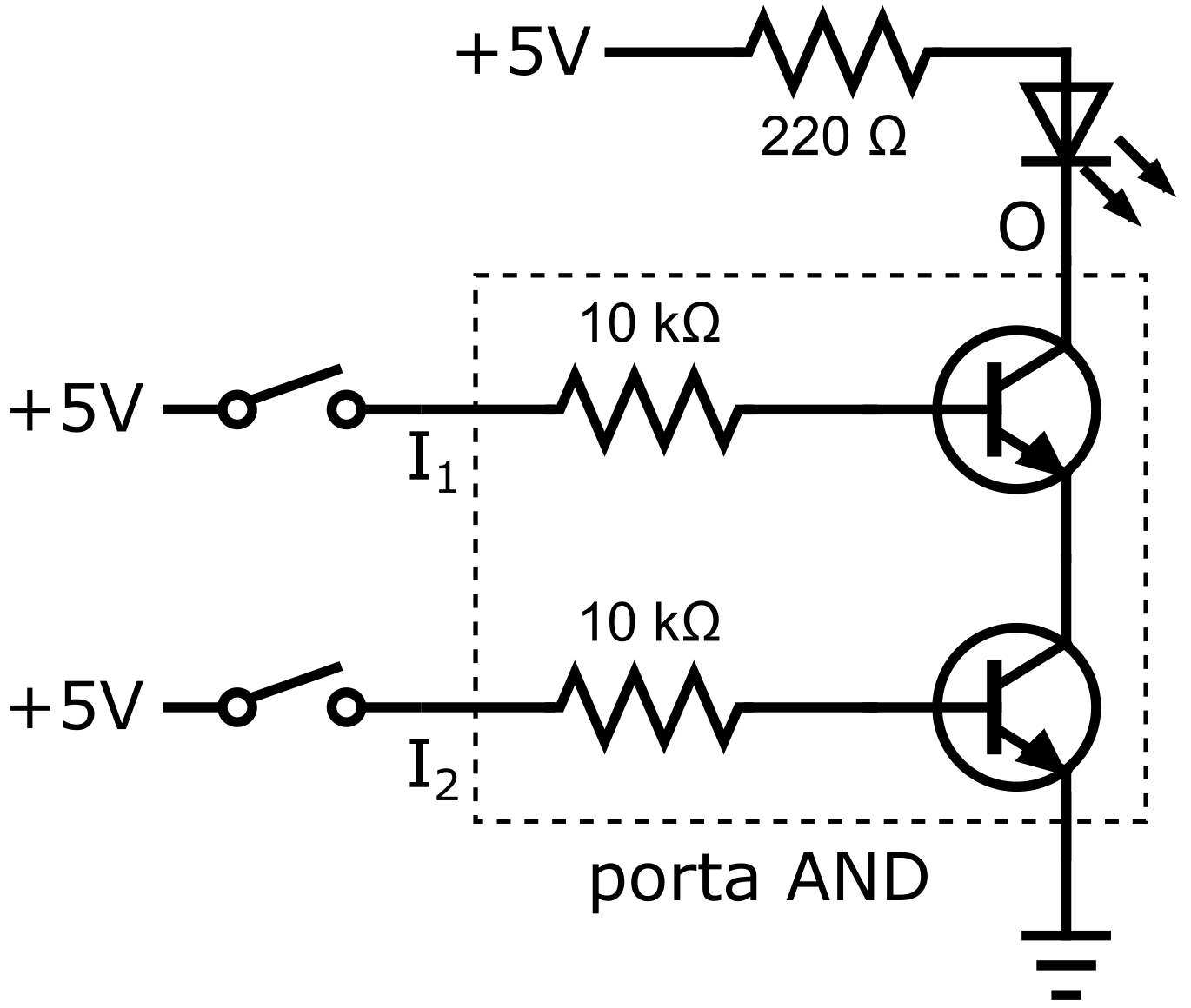
**Porta AND**

Una porta AND realizza un operatore binario che vale 1 se e solo se entrambi i propri argomenti sono 1 come illustrato nella tabella di verità di Figura 1.1.2.2.

| **X** | **Y** | **X AND Y** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

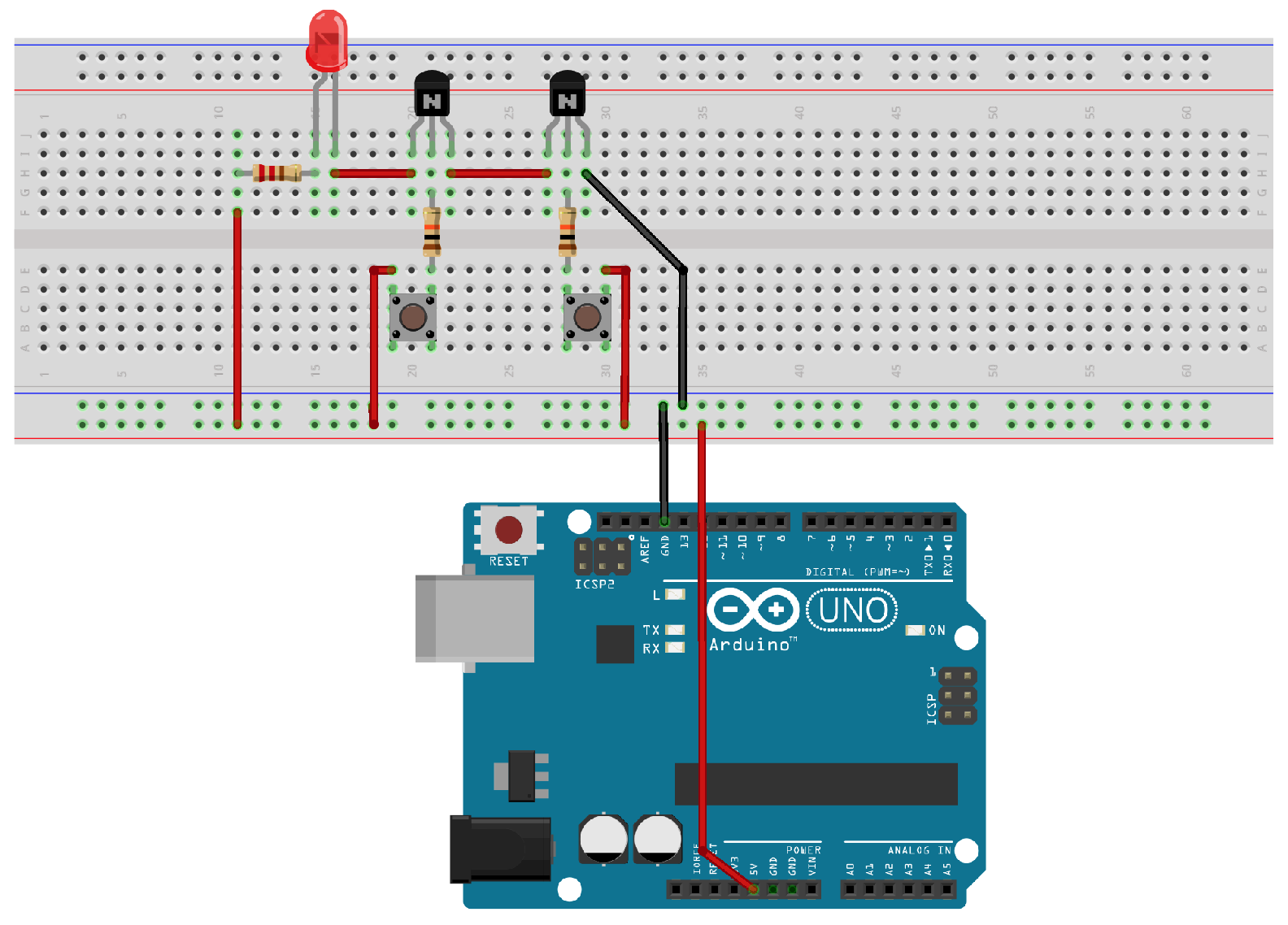
**Figura 1.1.2.2**: tabella di verità dell'operatore AND

Una possibile realizzazione di una porta AND mediante transistor NPN è illustrata in Figura 1.1.2.3.



**Figura 1.1.2.3**: circuito schematico basato su transistor NPN che realizza una porta AND

Il circuito usa due transistor e accende il led O se e solo se entrambi gli interruttori I1 e I2 sono chiusi. Si noti che la corrente attraversa la resistenza da 220 Ohm e il led, andando a massa, se e solo le basi di entrambi i transistor sono raggiunti da corrente. Una demo pratica della logica AND usando breadboard è mostrata in Figura 1.1.2.4.



**Figura 1.1.2.4**: realizzazione di una porta AND su breadboard.

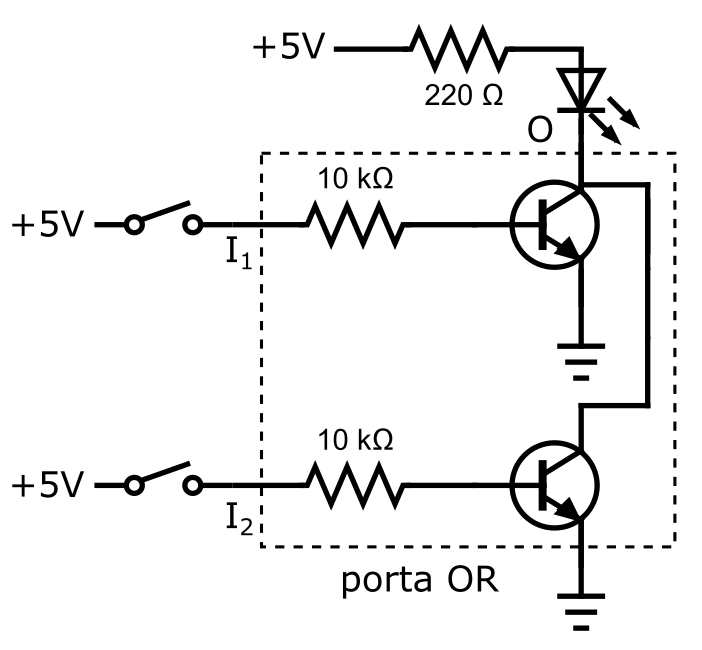
**Porta OR**

Una porta OR realizza un operatore binario che vale 1 se e solo se almeno uno dei propri argomenti sono 1 come illustrato nella tabella di verità di Figura 1.1.2.5.

| **X** | **Y** | **X OR Y** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

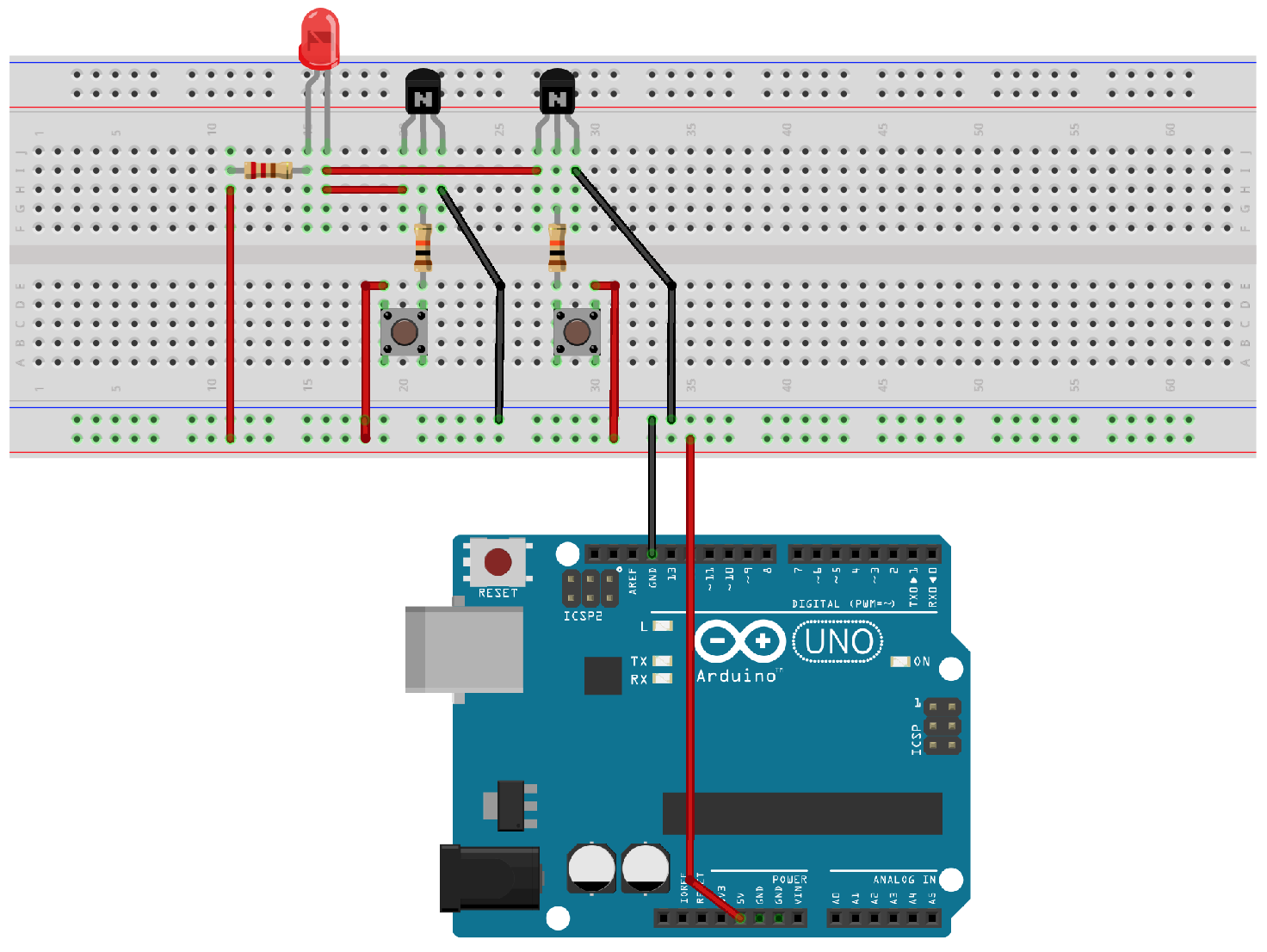
**Figura 1.1.2.5**: tabella di verità dell'operatore OR

Una possibile realizzazione di una porta OR mediante transistor NPN è illustrata in Figura 1.1.2.6.



**Figura 1.1.2.6**: circuito schematico che realizza una porta OR

Il circuito usa due transistor e accende il led O se e solo se almeno uno degli interruttori I1 e I2 è chiuso. Si noti che la corrente attraversa la resistenza da 220 Ohm e il led, andando a massa, se e solo se la base di almeno un transistor è raggiunta da corrente. Una demo pratica della logica OR usando breadboard è mostrata in Figura 1.1.2.7.



**Figura 1.1.2.7**: realizzazione di una porta OR su breadboard.

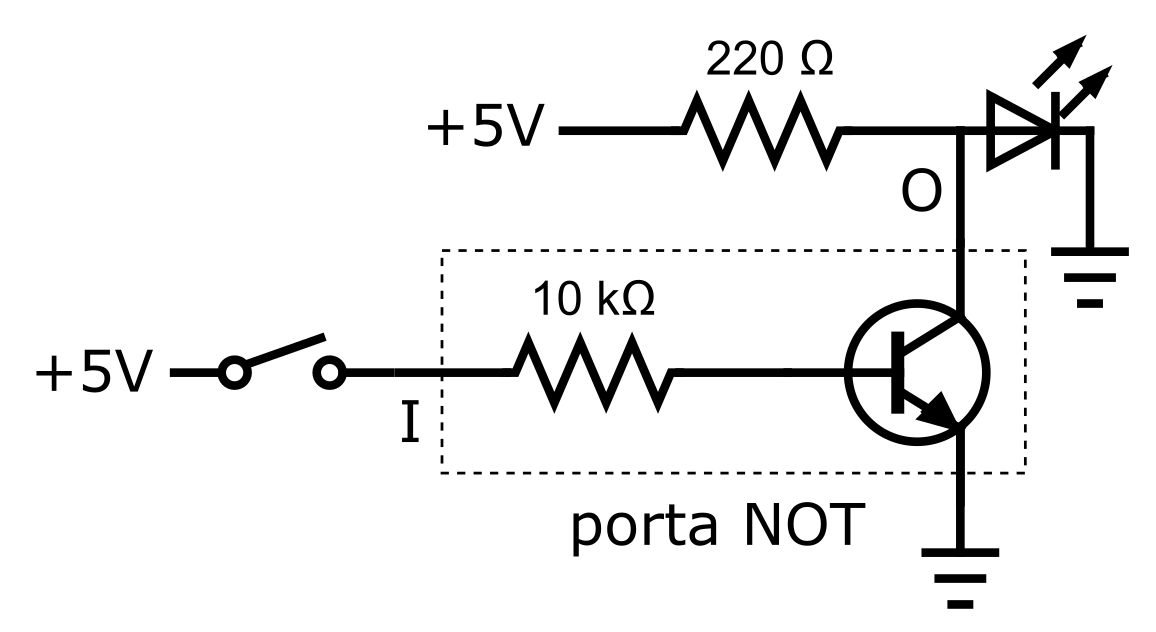
**Porta NOT**

Una porta NOT realizza un operatore unario che vale 1 se e solo se il proprio argomento è 0, invertendone il valore di verità come illustrato nella tabella di Figura 1.1.2.8.

| **X** | **NOT X** |
| --- | --- |
| 0 | 1 |
| 1 | 0 |

**Figura 1.1.2.8**: tabella di verità dell'operatore OR

Una possibile realizzazione di una porta NOT mediante transistor NPN è illustrata in Figura 1.1.2.9.

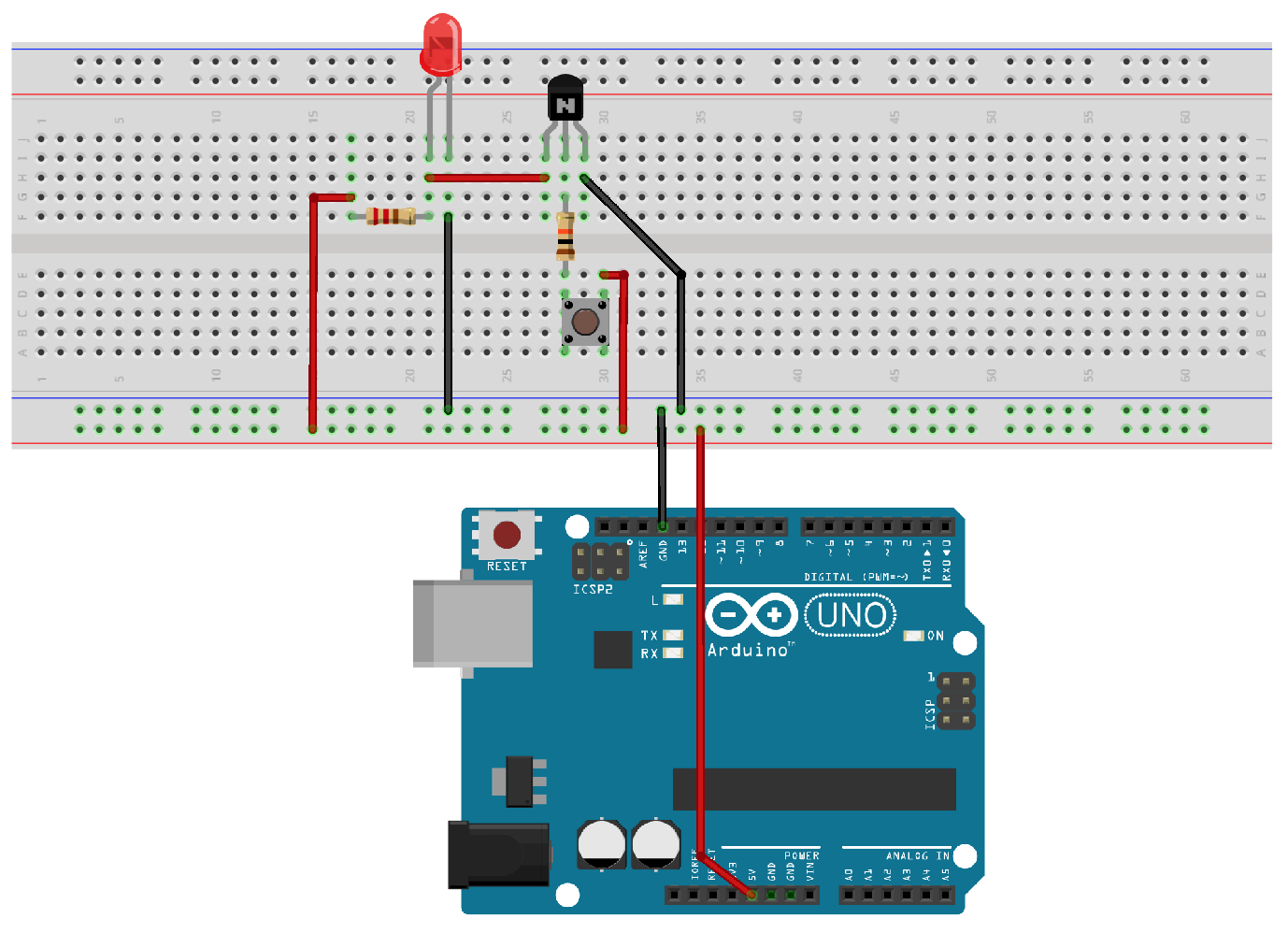


**Figura 1.1.2.9**: circuito schematico di una porta NOT.

Si noti che chiudendo l'interruttore la base viene raggiunta da corrente. Questo fa in modo che la corrente che attraversa la resistenza da 220 Ohm venga cortocircuitata a massa e non attraversi il led. Se invece l'interruttore è aperto, la corrente attraversa la resistenza da 220 Ohm e il led e va a massa senza attraversare il transistor. Una demo pratica della logica NOT usando breadboard è mostrata in Figura 1.1.2.10.

#### 1.1.2.2 Miniaturizzazione e VLSI

I circuiti visti finora sono realizzati usando transistor convenzionali. Le CPU reali utilizzano invece un procedimento di miniaturizzazione, chiamato Very Large Scale Integration (VLSI), che consente di realizzare circuiti visibili esclusivamente al microscopio. Un esempio è dato in Figura 1.1.2.11 che mostra un wafer al silicio (in basso) che contiene centinaia di chip, mostrati al microscopio (sopra). Attualmente, si è in grado di realizzare circa 17 milioni di transistor per millimetro quadrato, vale a dire circa 3 miliardi di transistor per una CPU di ultima generazione come un Intel core i7.



**Figura 1.1.2.10**: breadboard che realizza una porta NOT.

|  |  |
| --- | --- |
|  |  |

**Figura 1.1.2.11**: chip realizzato mediante tecnologia VLSI e wafer che contiene centinaia di chip

#### 1.1.2.3 Circuiti combinatori

E' possibile combinare le porte logiche viste precedentemente per costruire circuiti più complessi, noti come **circuiti combinatori**, che realizzano funzioni booleane arbitrarie. In un'espressione, gli operatori hanno priorità, dalla maggiore alla minore: NOT, AND, OR. Pertanto, NOT X OR X AND Y equivale a (NOT X) OR (X AND Y).

**Esempio (addizionatore parziale)**: mostriamo come realizzare un addizionatore a 1 bit con riporto. La funzione calcolata ha due entrate (i bit da sommare) e due uscite (la somma e il riporto) come illustrato nella tabella di verità di Figura 1.1.2.12.

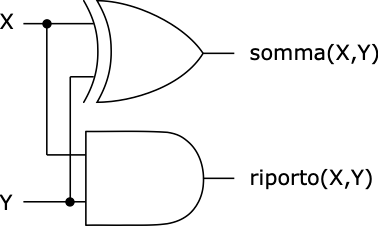
| **X** | **Y** | **somma(X,Y)** | **riporto(X,Y)** |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

**Figura 1.1.2.12**: tabella di verità addizionatore

Considerando le righe dove il risultato è 1, possiamo scrivere la funzione booleana da calcolare come segue:

* somma(X,Y) = (NOT X AND Y) OR (X AND NOT Y) = X XOR Y
* riporto(X,Y) = X AND Y

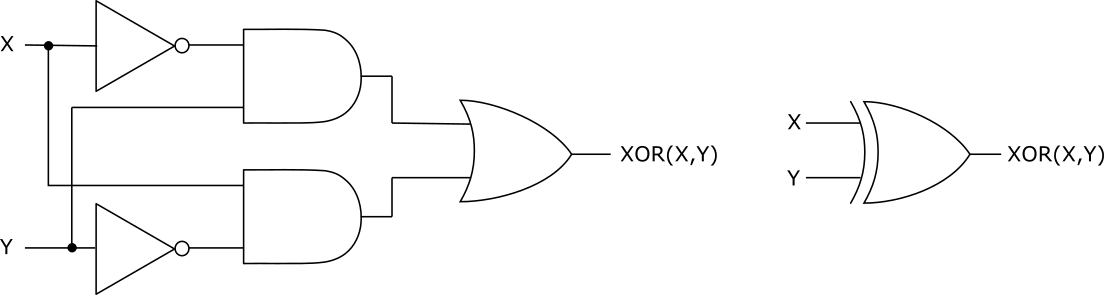
Dove l'operatore XOR, anche noto come OR esclusivo, è caratterizzato della tabella di verità di Figura 1.1.2.14. In Figura 1.1.2.13 mostriamo un circuito combinatorio che realizza l'addizionatore. La porta XOR può essere realizzata con due porte AND e due NOT come illustrato in Figura 1.1.2.15.



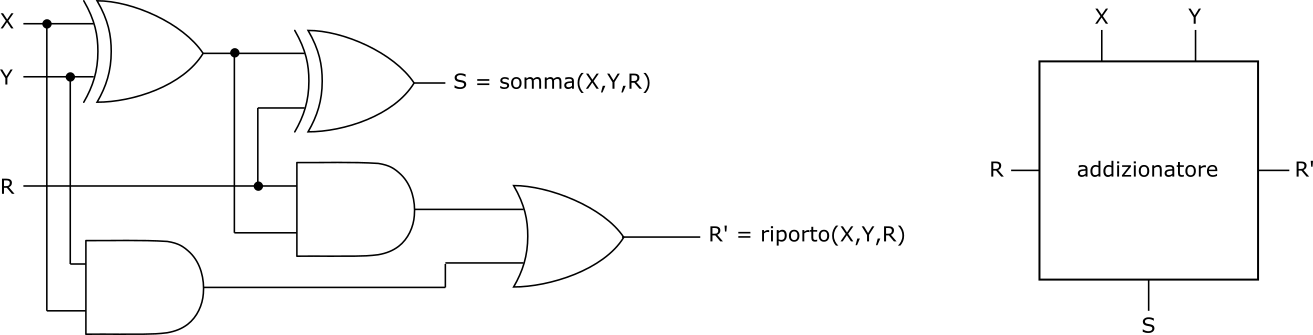
**Figura 1.1.2.13**: addizionatore binario con riporto che usa una porta XOR e una AND

| **X** | **Y** | **X XOR Y** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Figura 1.1.2.14**: tabella di verità dell'operatore XOR

  
  
**Figura 1.1.2.15**: definizione della porta XOR in termini di porte AND e NOT

**Esempio (addizionatore completo)**: vediamo ora come costruire un circuito combinatorio, l'addizionatore completo, che consente di costruire un addizionatore a n bit. La differenza con l'addizionatore parziale è che l'addizionatore completo prende in input, oltre ai due bit da sommare, anche un bit di riporto. Come vedremo, questo bit di riporto è generato da un addizionatore completo precedente in una catena di addizionatori bit a bit che lavorano in cascata. In Figura 1.1.2.16 mostriamo un circuito combinatorio che realizza un addizionatore completo e un suo packaging come elemento a "scatola nera". Figura 1.1.2.17 mostra la sua tabella di verità.

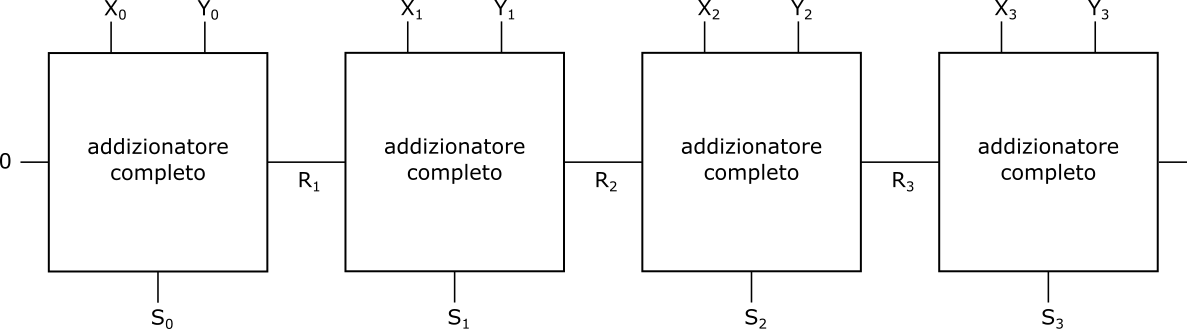


**Figura 1.1.2.16**: circuito combinatorio che realizza un addizionatore completo

| **X** | **Y** | **R** |  | **S** | **R'** |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 |  | 0 | 0 |
| 0 | 0 | 1 |  | 1 | 0 |
| 0 | 1 | 0 |  | 1 | 0 |
| 0 | 1 | 1 |  | 0 | 1 |
| 1 | 0 | 0 |  | 1 | 0 |
| 1 | 0 | 1 |  | 0 | 1 |
| 1 | 1 | 0 |  | 0 | 1 |
| 1 | 1 | 1 |  | 1 | 1 |

**Figura 1.1.2.17**: tabella di verità di un addizionatore completo

**Esempio (addizionatore a n bit)**: completiamo la trattazione dei circuiti combinatori mostrando come concatenare in cascata n istanze di un addizionatore completo a formare un addizionatore a n bit. Il circuito risultante per n=4 è illustrato in Figura 1.1.2.18.



**Figura 1.1.2.18**: addizionatore a 4 bit

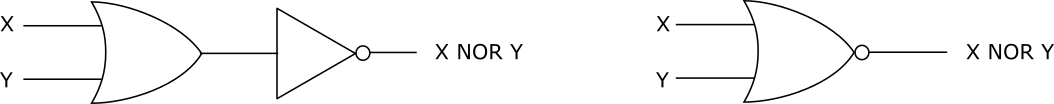
L'addizionatore mostrato calcola la somma di due numeri a 4 bit X3X2X1X0 + Y3Y2Y1Y0 = S3S2S1S0.

#### 1.1.2.4 Logica sequenziale e flip-flop

Diversamente dai circuiti combinatori, che realizzano funzioni booleane in cui valore di output dipende esclusivamente dal valore attuale delle sue linee di input, i circuiti sequenziali hanno una **nozione di stato**, ovvero una memoria di eventi accaduti nel passato. La realizzazione tipica usata nei calcolatori elettronici è **sincrona** e prevede un temporizzatore (**clock**) che invia impulsi in cui un segnale elettrico oscilla tra acceso e spento con una determinata frequenza[[2]](#footnote-1). Il segnale si propaga in un circuito elettronico a velocità variabili, che dipendono da vari fattori inclusa la temperatura[[3]](#footnote-2). Componenti diversi potrebbero avere tempi di propagazione diversi. Per questo e altri motivi, come la dissipazione di calore, **la frequenza di clock di una CPU non può superare una certa soglia** salvo generare instabilità e inaffidabilità nel sistema.

Un classico esempio di circuito in logica sequenziale è il **flip-flop**, un componente costruito mediante porte logiche che funge da mattoncino per costruire memorie ad accesso veloce come i **registri** della CPU. Vi sono diverse varianti di flip-flop. Noi vedremo il più comune, il flip-flop SR, che ha due ingressi S ed R e due uscite Q e Q': un segnale inviato all'ingresso S (set) imposta a 1 l'uscita Q, mentre uno inviato su R (reset) la imposta a 0. Si ha Q' = NOT Q. Se nessun segnale viene inviato su S e su R, il flip-flop mantiene l'ultimo stato in cui è stato posto. Se viene inviato un segnale simultaneamente su S e su R, lo stato del flip-flop diviene indefinito. Di per sè un flip-flop SR è asincrono, ma se sollecitato da un oscillatore si comporta in modo sincrono e può cambiare stato solo durante una pulsazione.

Per realizzare un flip-flop SR useremo una porta NOR, che è semplicemente una porta OR con l'output invertito, come illustrato in Figura 1.1.2.19. La tabella di verità dell'operatore NOR è mostrata in Figura 1.1.2.19.



**Figura 1.1.2.19**: definizione della porta NOR

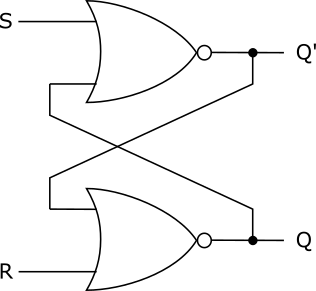
Il flip-flop SR, mostrato in Figura 1.1.2.20, è basato sulla tabella di transizione di Figura 1.1.2.19. In letteratura questo tipo di flip-flop viene detto anche **latch**, ovvero un dispositivo elettronico che mantiene il suo stato fino a quando non viene sollecitato con cambiamenti del suo input (set/reset).

| **X** | **Y** | **X NOR Y** |
| --- | --- | --- |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**Figura 1.1.2.19**: tabella di verità dell'operatore NOR

| **S** | **R** |  | **Q** | **Q'** |
| --- | --- | --- | --- | --- |
| 0 | 0 |  | stato immutato | |
| 0 | 1 |  | 0 | 1 |
| 1 | 0 |  | 1 | 0 |
| 1 | 1 |  | indefinito | |

**Figura 1.1.2.19**: tabella di transizione di un flip-flop SR



**Figura 1.1.2.20**: flip-flop (latch) SR

Si noti come la stabilità sia garantita dal meccanismo di feedback in cui l'output di una porta NOR è portata come input dell'altra porta NOR. Consideriamo ad esempio S=1 e R=0. Poiché una porta NOR manda in output 1 se e solo se entrambi gli input sono zero, Q' vale 0. Pertanto gli input della seconda porta (Q' ed R) sono 0, il che porta a Q=1. A questo punto, anche se S smette di essere 1, Q' rimane a 0 e Q rimane a 1.

## 1.2 Sistema operativo

I sistemi di calcolo più complessi si basano su sistemi operativi, ovvero software sviluppati per consentire l'esecuzione di applicazioni (programmi) sull'hardware disponibile. In questa dispensa faremo l'assunzione di trattare con un sistema operativo della famiglia Linux su hardware x86.

### 1.2.1 Compiti

I compiti principali di un sistema operativo sono:

* **gestione dei processi,** ovvero dei programmi in esecuzione (creazione, eliminazione, assegnazione di tempo della CPU, ecc.)
* **allocazione delle risorse ai processi**, come memoria principale (RAM), secondaria (dischi), e connessioni di rete
* misurazione e gestione del **tempo** (orologio, timer, ecc.)
* gestione delle **utenze** per consentire a più utenti di usare il sistema mediante l'assegnazione di **privilegi** di accesso alle risorse disponibili (es. file, programmi, ecc.)
* **comunicazione** tra processi che girano in locale sulla stessa macchina o su sistemi di calcolo distinti connessi da una rete
* **gestione degli errori** che possono emergere durante l'esecuzione dei processi o del sistema stesso
* **protezione e sicurezza**, **isolando** i processi affinché non possano interferire con altri processi, con il sistema operativo o con l'hardware, ad esempio esfiltrando dati o manomettendone il funzionamento in modo **malevolo** o per via di errori di programmazione (**bug**).

### 1.2.2 Obiettivi di progettazione

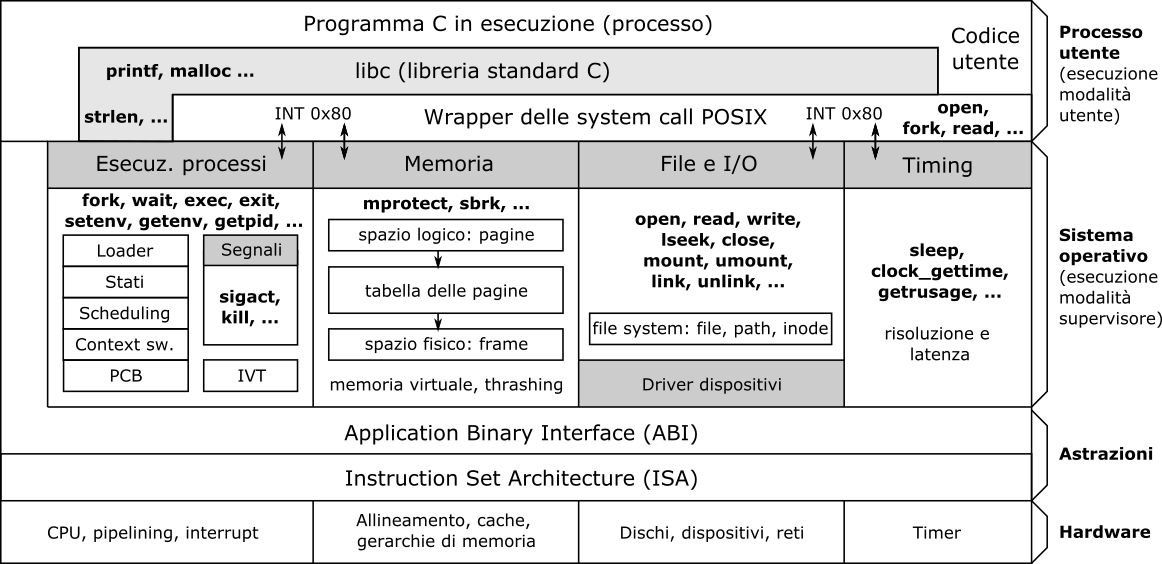
I sistemi operativi moderni sono costruiti con diversi obiettivi di progettazione in mente, fra cui:

* **multitasking**, ovvero la possibilità di avere varie applicazioni simultaneamente in esecuzione
* **preemption**, ovvero la possibilità di togliere il controllo della CPU a un processo per darlo a un altro processo
* **progresso** e assenza di **starvation**, ovvero l'assegnazione periodica della CPU ai processi per consentirne il progresso della computazione
* **alto throughput**, ovvero massimizzazione del numero di istruzioni eseguite dal sistema nell'unità di tempo.
* **bassa latenza**, ovvero minimizzazione del tempo richiesto dalle singole operazioni svolte dal sistema operativo. Questo è particolarmente importante in **ambienti real-time**, dove il sistema deve rispondere rapidamente alle sollecitazioni esterne (si pensi ad esempio a un software automobilistico che deve controllare l'apertura di un airbag in caso di collisione).

### 1.2.3 Panoramica del funzionamento

La Figura 1.2 illustra gli aspetti principali del sistema che tratteremo in questa dispensa, dalle applicazioni a scendere fino all'hardware:

1) In alto abbiamo i processi, che hanno la possibilità di richiamare funzionalità offerte da **librerie software**, come quelle standard del linguaggio C, oppure chiedere servizi direttamente al sistema operativo invocando funzioni wrapper[[4]](#footnote-3) che passano il controllo al sistema operativo effettuando chiamate a sistema (**system call**). Si noti che parte delle funzioni libreria C richiamano a loro volta system call, come printf, mentre altre no, come strlen. Nel loro insieme, tratteremo le librerie secondo lo standard [POSIX](https://pubs.opengroup.org/onlinepubs/9699919799/), che include le librerie standard del linguaggio C (ISO C)[[5]](#footnote-4).



**Figura 1.2**: panoramica di un sistema di calcolo Linux/x86 come trattato nella dispensa.

2) Programmi e librerie aderiscono a un insieme di **convenzioni** definite da una particolare Application Binary Interface (**ABI**), che definisce ad esempio il modo in cui i parametri vengono passati a una funzione e come questa restituisce un valore al chiamante.

3) Un programma eseguibile per la piattaforma usata si basa inoltre su una specifica Instruction Set Architecture (**ISA**), che definisce lo stato della CPU, le sue istruzioni, e l'effetto delle istruzioni sullo stato.

4) I programmi girano normalmente in una modalità a privilegi ridotti, detta **modalità utente**, che impedisce loro di effettuare operazioni potenzialmente pericolose per il sistema. Quando un processo esegue codice del sistema operativo a seguito di una system call, passa in **modalità supervisore** (o sistema, o kernel), acquisendo privilegi completi nell'esecuzione della system call per poter avere il pieno controllo della macchina.

5) Il passaggio del controllo di un programma al sistema operativo (ovvero al suo kernel, o nucleo) per effettuare **chiamate a sistema** avviene normalmente mediante il meccanismo degli **interrupt**. Un interrupt è un evento che interrompe il flusso di esecuzione del programma e passa il controllo a codice del sistema operativo, cambiando modalità di esecuzione in modo che il programma abbia privilegi più alti. Un interrupt può essere generato **dalla CPU stessa** (es. mediante l'istruzione INT dei processori x86 o una condizione di errore della CPU), oppure da **eventi esterni** come la pressione di un tasto sulla tastiera.

6) I **moduli** principali del kernel del sistema operativo, attivati mediante interrupt, sono adibiti alla gestione dei processi, della memoria, di file e I/O in genere, e del tempo.

6a) Il modulo di **esecuzione dei processi** è il più importante ed è adibito al caricamento di un programma eseguibile (**loading**), alla sua **schedulazione** (assegnazione della CPU), alla **gestione degli interrupt**, al mantenimento delle informazioni sull'esecuzione in una struttura chiamata Process Control Block (**PCB**) e alla gestione dei **segnali**, un meccanismo di notifica di eventi tra processi e tra sistema operativo e processi (in ambiente Unix-like come Linux e MacOS X).

6b) Il modulo di **gestione della memoria** fornisce nei sistemi operativi moderni un meccanismo di **virtualizzazione della memoria** in modo che ogni processo veda una porzione tutta sua e non sia possibile per un processo accedere alla memoria di un altro processo senza che quest'ultimo ne sia a conoscenza. Questo meccanismo di **protezione** impedisce ad esempio a un programma malevolo o difettoso di accedere allo spazio di memoria di un altro programma alternandone il contenuto. La virtualizzazione è basata su una **separazione tra memoria fisica e memoria logica** accessibile ai programmi. Un altro aspetto della gestione della memoria riguarda l'**allocazione dello spazio logico** disponibile ai programmi stessi, ad esempio in C usando malloc e free.

6c) Il modulo di **gestione dei file e dell'I/O** in generale si occupa di soddisfare le richieste di uso dei dischi e di dispositivi da parte dei programmi. Particolarmente importante sono i **driver**, componenti software che forniscono un'interfaccia standard verso il kernel astraendo le caratteristiche specifiche dei dispositivi esterni. Essi possono essere installati e aggiornati modularmente in un secondo tempo e non fanno parte del kernel del sistema operativo.

6d) Il modulo di **gestione del tempo** fornisce al kernel e alle applicazioni funzionalità per la misurazione del tempo reale e del tempo speso attivamente sulla CPU su base singolo processo.

7) L'uso dell'**hardware soggiacente** da parte di un programma avviene come segue:

7a) la CPU può essere usata dal programma in modalità utente **senza il tramite del sistema operativo**. Questo avviene quando il programma gira in **modalità utente**

7b) ogni altra interazione con l'hardware è mediata dal sistema operativo.

## 

## 

## 1.3 Domande riepilogative

Le seguenti domande riepilogative sugli argomenti del capitolo ammettono come risposte **vero** o **falso**:

1. I router che connettono le reti locali domestiche a Internet possono essere considerati sistemi di calcolo?
2. In una macchina di Von Neumann, programmi e dati risiedono in memorie fisiche separate?
3. I bus sono strutture di interconnessione che connettono le varie componenti di un sistema di calcolo?
4. L'unità aritmetico-logica (ALU) è un componente esterno alla CPU che consente di eseguire operazioni come somme, prodotti, ecc.?
5. L'I/O bridge si occupa esclusivamente di coordinare lo scambio dei dati fra la memoria e i dispositivi di I/O?
6. Gli adattatori sono saldati sulla scheda madre?
7. Il sistema di memoria di un sistema di calcolo complesso come un laptop è costituito da vari livelli che includono memorie cache veloci, delegate a mantenere il contenuto delle celle di memoria più frequentemente usate o adiacenti.?
8. Le porte logiche sono gli elementi base con cui sono costruiti i transistor?
9. Le principali porte logiche sono AND, OR e NOT?
10. E' possibile costruire una porta logica XOR in termini delle porte logiche AND, OR e NOT?
11. Tutte le seguenti affermazioni sono vere? 0 AND 1 = 0, NOT 1 = 0, 1 OR 0 = 1, 1 XOR 1 = 0, 1 AND 1 = 1, 1 XOR 0 = 1, 0 XOR 0 = 0?
12. I transistor sono componenti elettronici basati su semiconduttori il cui scopo è esclusivamente quello di amplificare un segnale elettrico?
13. Le CPU di ultima generazione contengono milioni di transistor?
14. Un wafer (non l'alimento ☺) è un supporto semiconduttore circolare che contiene centinaia di chip che una volta ritagliati dal wafer formano circuiti integrati come CPU o memorie?
15. I circuiti combinatori calcolano funzioni dove i segnali di output sono esclusivamente funzione dei segnali di input?
16. I circuiti sequenziali sono come i circuiti combinatori, con l'unica differenza che possono inviare segnali in input sequenzialmente e non in parallelo?
17. Un flip-flop è un circuito combinatorio?
18. Un addizionatore con riporto può essere realizzato con una porta XOR e una porta AND?
19. Un addizionatore è un circuito combinatorio?
20. La frequenza di clock di una CPU può crescere indefinitamente al crescere del numero di transistor nella CPU?
21. Un flip-flop ha un segnale di ingresso e due di uscita?
22. In un flip-flop, solo uno dei segnali di output può essere 1?
23. Un sistema operativo si occupa della gestione dei processi?
24. Le nozioni di programma e processo sono equivalenti?
25. La gestione della memoria non viene effettuata con il tramite del sistema operativo?
26. La comunicazione fra processi è gestita interamente dall'hardware senza il tramite del sistema operativo?
27. Fra gli obiettivi di progetto di un sistema operativo vi è la massimizzazione del throughput?
28. Un processo è in starvation se ha richiesto una risorsa, ma è in attesa indefinita che quella risorsa gli venga assegnata dal sistema operativo?
29. Il progresso di un processo misura il numero di operazioni di I/O effettuate?
30. Una system call (chiamata a sistema) può essere effettuata direttamente da un programma utente oppure da una funzione di libreria?
31. Un'application binary interface (ABI) definisce lo stato della CPU, le sue istruzioni, e l'effetto delle istruzioni sullo stato?
32. La separazione tra modalità utente e supervisore consente di evitare che un processo, che gira in modalità utente, possa compromettere il sistema, delegando le operazioni potenzialmente pericolose al sistema operativo, che gira in modalità supervisore?
33. Un interrupt non può essere generato dalla CPU stessa, ma solo da un dispositivo esterno?
34. La CPU può essere usata dal programma in modalità utente senza il tramite del sistema operativo?

# 

# 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[[6]](#footnote-5)** 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**).

I programmatori professionisti interessati a scrivere codice particolarmente ottimizzato per le prestazioni oppure parti di sistemi operativi usano il linguaggio **assembly** (o assemblativo), 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. 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. | Vedi interpretazione. |

## 2.3 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 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.

### 

### 2.3.1. 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 redirezione > 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. |

### 2.3.2 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](http://en.wikipedia.org/wiki/Executable_and_Linkable_Format) 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](http://en.wikipedia.org/wiki/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 utilizzata[[7]](#footnote-6).  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 Linux[[8]](#footnote-7). Tutto questo indipendentemente dall’hardware soggiacente, che potrebbe anche essere lo stesso. Pertanto, **i formati eseguibili non garantiscono in genere la portabilità** dei programmi. |

### 2.3.3 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:   | $ 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):   |  | | --- |   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. |

## 2.4 Domande riepilogative

Le seguenti domande riepilogative sugli argomenti del capitolo ammettono come risposte **vero** o **falso**:

1. Linguaggio macchina e assembly sono sinonimi?
2. I linguaggi di basso livello sono più vicini all'hardware e hanno costrutti più semplici rispetto a quelli di alto livello?
3. L'output di un preprocessore C è un programma C?
4. Il modulo compilatore di gcc produce direttamente codice in linguaggio macchina?
5. Il compito di un assemblatore è quello di tradurre in linguaggio macchina un programma assembly?
6. File oggetto e file eseguibile sono la stessa cosa?
7. Il linker costituisce l'ultimo stadio della pipeline di compilazione e si occupa di collegare i simboli (nomi funzioni, nomi variabili, ecc.) che appaiono nel programma con indirizzi di memoria?
8. Il comando file restituisce l'estensione del file a cui viene applicato?
9. Il tool objdump -d file disassembla il file oggetto o eseguibile?
10. Il tool objdump -S produce un disassemblato dove appare solo la versione assembly e non quella binaria?

# 

# 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:
  1. lo stato della CPU;
  2. il formato delle sue istruzioni;
  3. 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 e nel [Paragrafo 3.3](#_uzsk4ytkbvs6) mostreremo le principali differenze con x86-64.

## 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 | *non supportato* | - | 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, SI, BP e SP, 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, Mb[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 | (Ebase) | M[R[Ebase]] | Indiretto |
| Memoria | imm(Ebase) | M[imm+R[Ebase]] | Base e spiazzamento |
| Memoria | (Ebase,Eindice) | M[R[Ebase]+R[Eindice]] | Base e indice |
| Memoria | imm(Ebase,Eindice) | M[imm+R[Ebase]+R[Eindice]] | Base, indice e spiazzamento |
| Memoria | (Eindice,s) | M[R[Eindice]∙s] | Indice e scala |
| Memoria | imm(Eindice,s) | M[imm+R[Eindice]∙s] | Indice, scala e spiazzamento |
| Memoria | (Ebase,Eindice,s) | M[R[Ebase]+R[Eindice]∙s] | Base, indice e scala |
| Memoria | imm(Ebase,Eindice,s) | M[imm+R[Ebase]+R[Eindice]∙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 *Sn* per denotare un operando **sorgente** di n byte, e *Dn* 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** |
| --- | --- | --- |
| *MOV S,D* | D ← S | *copia byte da sorgente S a destinazione D* |
| movb S1,D1 | D1 ← S1 | copia 1 byte |
| movw S2,D2 | D2 ← S2 | copia 2 byte |
| movl S4,D4 | D4 ← S4 | 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** |
| --- | --- | --- |
| *MOVZ S,D* | D ← ZeroExtend(S) | *copia byte da sorgente S a destinazione D, riempiendo con zero i byte che D ha in più rispetto a S* |
| movzbw S1,D2 | D2 ← 0x00:S1 | copia 1 byte in 2 byte, estendi con zero |
| movzbl S1,D4 | D4 ← 0x000000:S1 | copia 1 byte in 4 byte, estendi con zero |
| movzwl S2,D4 | D4 ← 0x0000:S2 | copia 2 byte in 4 byte, estendi con zero |

**Esempi:**

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

| **Istruzione** | **Risultato** (estensione sottolineata) |
| --- | --- |
| movzbw %al,%cx | %eax=0x123412**34** %ecx=0xABCD**0034** |
| movzbl %al,%ecx | %eax=0x123412**34** %ecx=0x**00000034** |
| movzwl %ax,%ecx | %eax=0x1234**1234** %ecx=0x**00001234** |

Vediamo ora l’istruzione MOVS:

| **Istruzione** | **Effetto** | **Descrizione** |
| --- | --- | --- |
| *MOVS S,D* | D ← SignExtend(S) | *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* |
| movsbw S1,D2 | D2 ← 0xMM:S1 | copia 1 byte in 2 byte, estendi con M=’F’ se il bit più significativo di S1 (bit del segno) è 1 e con M=’0’ altrimenti |
| movsbl S1,D4 | D4 ← 0xMMMMMM:S1 | copia 1 byte in 4 byte, estendi con M=’F’ se il bit più significativo di S1 (bit del segno) è 1 e con M=’0’ altrimenti |
| movswl S2,D4 | D4 ← 0xMMMM:S2 | copia 2 byte in 4 byte, estendi con M=’F’ se il bit più significativo di S2 (bit del segno) è 1 e con M=’0’ altrimenti |

**Esempi:**

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

| **Istruzione** | **Risultato** (estensione sottolineata) |
| --- | --- |
| movsbw %al,%cx | %eax=0x123412**34** %ecx=0xABCD**0034** |
| movsbl %al,%ecx | %eax=0x123412**34** %ecx=0x**00000034** |
| movswl %ax,%ecx | %eax=0x1234**1234** %ecx=0x**00001234** |
| movsbw %cl,%ax | %ecx=0xABCDE1**E2** %eax=0x1234**FFE2** |
| movsbl %cl,%eax | %ecx=0xABCDE1**E2** %eax=0x**FFFFFFE2** |
| movswl %cx,%eax | %ecx=0xABCD**E1E2** %eax=0x**FFFFE1E2** |

##### **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** |
| --- | --- | --- |
| pushl S4 | R[%esp] ← R[%esp] - 4 M[R[%esp]] ← S4 | copia l’operando di 4 byte S sulla cima della stack |
| popl D4 | D4 ← M[R[%esp]] R[%esp] ← 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← D+1 | incrementa destinazione |
| DEC D | D← D-1 | decrementa destinazione |
| NEG D | D← -D | inverti segno destinazione |
| NOT D | D← ~D | complementa a 1 destinazione |
| ADD S,D | D← D+S | aggiungi sorgente a destinazione e risultato in destinazione |
| SUB S,D | D← D-S | sottrai sorgente da destinazione e risultato in destinazione |
| IMUL S,D | D← D\*S | moltiplica sorgente con destinazione con segno e risultato in destinazione, **la destinazione deve essere un registro** |
| XOR S,D | D← D^S | or esclusivo sorgente con destinazione e risultato in destinazione |
| OR S,D | D← D|S | or sorgente con destinazione e risultato in destinazione |
| AND S,D | D← 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 ← D:A / V  D ← 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[[9]](#footnote-8) V1 | %al ← %ax / V1  %ah ← %ax % V1 | calcola simultaneamente il quoziente e il resto della divisione del dividendo %ax con il divisore V1 che può essere registro o memoria a 8 bit[[10]](#footnote-9). |
| idivw V2 | %ax ← %dx:%ax / V2  %dx ← %dx:%ax % V2 | calcola simultaneamente il quoziente e il resto della divisione del dividendo ottenuto concatenando %dx con %ax, dove %dx ha i bit più significativi, con il divisore V2 che può essere un registro o memoria a 16 bit |
| idivl V4 | %eax ← %edx:%eax / V4  %edx ← %edx:%eax % V4 | calcola simultaneamente il quoziente e il resto della divisione del dividendo ottenuto concatenando %edx con %eax, dove %edx ha i bit più significativi, con il divisore V4 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,  sarl $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 ipoteca 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** |
| --- | --- | --- |
| leal S,D4 | D4 ← &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 M4[0x100]=0xCAFE

| **Istruzione** | **Effetto** | **Risultato** |
| --- | --- | --- |
| movl (%eax),%edx | R[%edx] ← M4[R[%eax]] | %edx=0xCAFE |
| leal (%eax),%edx | R[%edx] ← R[%eax] | %edx=0x100 |

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

| **Istruzione** | **Effetto** | **Risultato** |
| --- | --- | --- |
| leal (%eax,%ecx),%edx | R[%edx] ← R[%eax]+R[%ecx] | %edx=0x107 |
| leal -3(%eax,%ecx),%edx | R[%edx] ← R[%eax]+R[%ecx]-3 | %edx=0x104 |
| leal -3(%eax,%ecx,2),%edx | R[%edx] ← R[%eax]+R[%ecx]∙2-3 | %edx=0x10B |
| leal (%eax,%ecx,2),%edx | R[%edx] ← R[%eax]+R[%ecx]∙2 | %edx=0x10E |
| leal (%ecx,4),%edx | R[%edx] ← R[%ecx]∙4 | %edx=0x1C |

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 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** |
| --- | --- | --- |
| int x=10; // x è in %eax  int y=20; // y è in %ecx  int z=y; // z è in %edx  z=z\*4;  z=z-7;  z=z+x; | movl $10,%eax  movl $20,%ecx  movl %ecx,%edx  imull $4,%edx  addl $-7,%edx  addl %eax,%edx | movl $10,%eax  movl $20,%ecx  leal -7(%eax,%ecx,4),%edx |

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 ← D << X | effettua shift **logico** o **aritmetico** a **sinistra** dell'operando D di X posizioni dove:   * D è registro o memoria (da 1,2,4 byte) * X può essere %cl, imm, o 1 * se X=1, X può essere omesso |
| SHR *X,D* | D ← D >> X | effettua shift **logico** a **destra** dell'operando D di X posizioni dove:   * D è registro o memoria (da 1,2,4 byte) * X può essere %cl, imm, o 1 * se X=1, X può essere omesso |
| SAR *X,D* | D ← D >> X | effettua shift **aritmetico** a **destra** dell'operando D di X posizioni dove:   * D è registro o memoria (da 1,2,4 byte) * X può essere %cl, imm, o 1 * se X=1, X può essere omesso |

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 23=8 dell'operando %al assunto con segno:

| **Prima** | **Operando** | **Esadecimale** | **Binario** | **Decimale** |
| --- | --- | --- | --- | --- |
| sarb $3, %al | %al | CA | **1**1001010 | -54 |
| **Dopo** | F9 | **111**11001 | -7 = -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** |
| --- | --- | --- | --- | --- |
| sarb $3, %al | %al | 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** |
| --- | --- | --- | --- | --- |
| shrb $3, %al | %al | 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** |
| --- | --- | --- |
| jmp etichetta | R[%eip] ← indirizzo associato all’etichetta | salto diretto |
| jmp \*S | R[%eip] ← 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 ← 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** | **Sinonimo** | **Condizione[[11]](#footnote-10)** | **Significato** |
| --- | --- | --- | --- |
| e | z | ZF | Uguale (o zero) |
| ne | nz | ~ZF | Diverso (o non zero) |
| s |  | SF | Negativo |
| ns |  | ~SF | Non negativo |
| g | nle | ~(SF^OF)&~ZF | Maggiore (g=greater) con segno |
| ge | nl | ~(SF^OF) | Maggiore o uguale (ge=greater or equal) con segno |
| l | nge | SF^OF | Minore (l=less) con segno |
| le | ng | (SF^OF)|ZF | Minore o uguale con segno |
| a | nbe | ~CF&~ZF | Maggiore (a=above) senza segno |
| ae | nb | ~CF | Maggiore o uguale (ae=above or equal) senza segno |
| b | nae | CF | Minore (b=below) senza segno |
| be | na | CF|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:

| decl %eax  jz L | eax--;  if (eax == 0) goto L; |
| --- | --- |

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:

| subl %ebx, %eax  je L | temp = eax  eax = eax - ebx  if (temp == ebx) goto L |
| --- | --- |

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** | **Sinoni-mo** | **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:

| cmpl %ebx, %eax  jle L | if (eax <= ebx) goto L; |
| --- | --- |

La prima operazione calcola la differenza R[%eax]-R[%ebx]. La seconda salta se R[%eax]-R[%ebx] <= 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]←R[%esp]-4  M[R[%esp]]←R[%eip]  R[%eip]← 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]←M[R[%esp]]  R[%esp]←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:

| call f  imull $3, %eax  …  f: movl $2, %eax  ret | f();  eax = eax\*3;  …  void f() {  eax = 2;  } |
| --- | --- |

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** |
| --- | --- | --- |
| CMOVcc S,D | if (condizione) D←S | se la condizione associata al suffisso cc 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:

| cmpl %ecx, %eax  cmovgl %eax, %ecx | if (eax > ecx) ecx = eax; |
| --- | --- |

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** |
| --- | --- | --- |
| TEST S,D | calcola S & D | 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:

| testl %eax, %eax  jz L | cmpl $0, %eax  jz L | if (eax==0) goto L |
| --- | --- | --- |

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** |
| --- | --- | --- |
| SETcc D1 | D1←condizione | se la condizione associata al suffisso cc è verificata, scrive 1 in D1, altrimenti scrive 0 |

**Esempio.**

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

| cmpl $0, %eax  setl %dl | dl = (eax < 0) | |
| --- | --- | --- |

##### **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 | cmovge**b** %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 |
| IDIV | usa i registri D ed A e l'operando divisore non può essere immediato | idivl %edi # ok  idivl **$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** |
| --- | --- | --- |
| if (test) blocco  istruzione | if (!test) goto L;  blocco;  L: istruzione; | IA32(test)  Jcc L  IA32(blocco)  L: IA32(istruzione) |

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** |
| --- | --- | --- |
| if (a>b) c=10;  c++; | if (a<=b) goto L;  c=10;  L: c++; | cmpl %ebx, %eax  jbe L  movl $10, %ecx  L: incl %ecx |

Notiamo che il test a<=b su variabili senza segno viene realizzato calcolando la differenza R[%eax]-R[%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[[12]](#footnote-11)** | **Traduzione IA32** |
| --- | --- | --- |
| if (test) blocco1  else blocco2  istruzione | if (!test) goto E;  blocco1  goto F;  E: blocco2  F: istruzione; | IA32(test)  Jcc E  IA32(blocco1)  jmp F  E: IA32(blocco2)  F: IA32(istruzione) |

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** |
| --- | --- | --- |
| if (a<=b) c=10;  else c=20;  c++; | if (a>b) goto E;  c=10;  goto F;  E: c=20;  F: c++; | cmpl %ebx, %eax  jg E  movl $10, %ecx  jmp F  E: movl $20, %ecx  F: incl %ecx |

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[[13]](#footnote-12)** | **Traduzione IA32** |
| --- | --- | --- |
| while (test) blocco  istruzione | L:  if (!test) goto E;  blocco;  goto L;  E: istruzione; | L: IA32(test)  Jcc E  IA32(blocco)  jmp L  E: IA32(istruzione) |

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.**

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** |
| --- | --- | --- |
| a=1;  c=0;  while (a<=b) {  c+=a;  a++;  } | a=1;  c=0;  L:  if (a>b) goto E;  c+=a;  a++;  goto L;  E: | movl $1, %eax  movl $0, %ecx  L: cmpl %ebx, %eax  ja E  addl %eax, %ecx  incl %eax  jmp L  E: |

Il programma calcola in c la somma dei primi b interi, cioè c ← 1+2+3+...+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[[14]](#footnote-13)** | **Traduzione IA32** |
| --- | --- | --- |
| for (ini;test;agg)  blocco  istruzione | ini;  L:  if (!test) goto E;  blocco;  agg;  goto L;  E: istruzione | IA32(ini)  L: IA32(test)  Jcc E;  IA32(blocco)  IA32(agg)  jmp L  E: IA32(istruzione) |

**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** |
| --- | --- | --- |
| c=0;  for (a=1; a<=b; a++)  c+=a; | c=0;  a=1;  L:  if (a>b) goto E;  c+=a;  a++;  goto L;  E: | movl $0, %ecx  movl $1, %eax  L: cmpl %ebx, %eax  ja E  addl %eax, %ecx  incl %eax  jmp L  E: |

Il programma calcola in c la somma dei primi b interi, cioè c ← 1+2+3+...+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](https://www.uclibc.org/docs/psABI-i386.pdf) (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** |
| --- | --- |
| void f(){ void g(){ void h(){  g(); … …  h(); } }  } | f: g: h:  call g … …  call h ret ret  ret |

##### **3.2.3.1 Restituzione valore**

Per convenzione, valori scalari come interi e puntatori[[15]](#footnote-14) 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** |
| --- | --- | --- |
| int f(){  return 7+g();  }  int g(){  return 10;  } | int f(){  int tmp = g();  tmp += 7;  return tmp;  }  int g(){  int tmp = 10;  return tmp;  } | f:  call g  addl $7, %eax  ret  g:  movl $10, %eax  ret |

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 in Figura 3.2.1 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**.[[16]](#footnote-15) |
| --- |

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**  **test.c** | **Traduzione IA32 (con ebp):**  **gcc -S test.c** | **Traduzione IA32 (no ebp):**  **gcc -S -fomit-frame-pointer test.c** |
| --- | --- | --- |
| int f() {  return 10;  } | f: pushl %ebp  movl %esp, %ebp  movl $10, %eax  popl %ebp  ret | f:  movl $10, %eax  ret |



**Figura 3.2.1**: stack frame

##### **3.2.3.3 Passaggio dei parametri**

I parametri di tipi primitivi[[17]](#footnote-16) 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.



**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 chiamante (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)** |
| --- | --- | --- |
| int f() {  return g()+h();  } | int f(){  int c = g();  int a = h();  a += c;  return a;  } | f:  call g  movl %eax, %ecx  pushl %ecx  call h  popl %ecx  addl %ecx, %eax  ret |

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)** |
| --- | --- | --- |
| int f() {  return g()+h();  } | int f(){  int b = g();  int a = h();  a += b;  return a;  } | f: pushl %ebx # prologo  call g  movl %eax, %ebx  call h  addl %ebx, %eax  popl %ebx # epilogo  ret |

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
3. se la funzione usa l’operatore & su una variabile locale, che quindi deve possedere 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)** |
| --- | --- | --- |
| int f(int x){  int y;  leggi(&y);  return x+y;  } | int f(int x){  int y;  int\* c=&y;  leggi(c);  int a = x;  a += y;  return a;  } | 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 |

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)** |
| --- | --- | --- |
| int f(int x){  int y;  leggi(&y);  return x+y;  } | int f(int x){  int y;  int\* c=&y;  leggi(c);  int a = x;  a += y;  return a;  } | 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 |

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 8 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=sizeof(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 l'operando memoria:

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** |
| --- | --- | --- |
| a[c]=10; | \*(a+c)=10; | movl $10, (%eax,%ecx,4) |

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 ebp)** |
| --- | --- | --- |
| int sum(int c[2]) {  return c[0]+c[1];  } | int sum(int c[2]){  int a = c[0];  a += c[1];  return a;  } | sum:  movl 4(%esp), %ecx  movl (%ecx), %eax  addl 4(%ecx), %eax  ret |

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)** |
| --- | --- | --- |
| int sum(int\* v, int n){  int i;  int s=0;  for (i=0; i<n; i++)  s += v[i];  return s;  } | 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;  } | 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 |

### 3.2.5 Strutture

In questo paragrafo vediamo come accedere ai campi di una struttura mediante puntatore e come gestire strutture allocate in stack come variabili locali. Assegnamento, passaggio di parametro e valore di ritorno di intere strutture è al di là degli scopi della dispensa. E' importante comprendere il layout di memoria di una struttura: si rimanda al [Paragrafo 5.2.3.3](#_1fvnl910x3g7) per una descrizione approfondita.

L'accesso a un campo di una struttura è possibile tramite l'operando memoria: off(base).

dove base è l’indirizzo del primo byte della struttura e off è lo spiazzamento in byte del campo rispetto alla base della struttura stessa.

**Esempio 1**. Si consideri l'accesso ai campi di una struttura mediante puntatore.

| **C da tradurre** | **Traduzione IA32** |
| --- | --- |
| struct person {  char age;  char\* name;  };  char get\_age(struct person \*p){  return p->age;  }  char \*get\_name(struct person \*p){  return p->name;  } | .globl get\_age  .globl get\_name  get\_age:  movl 4(%esp), %eax  movb **(%eax)**, %al  ret  get\_name:  movl 4(%esp), %eax  movl **4(%eax)**, %eax  ret |

Si noti in grassetto l'operando memoria per accedere ai due campi della struttura: l'offset del campo name è 4 poiché vi sono 3 byte di padding tra age e name, essendo name un campo a 32 bit.  
  
**Esempio 2**. Si consideri l'allocazione in stack di una struttura e la sua inizializzazione.

| **C da tradurre** | **Traduzione IA32** |
| --- | --- |
| struct person {  char age;  char\* name;  };  void f() {  struct person p;    fill(&p);  ...  } | .globl f  f:  subl $12, %esp # spazio in stack per p  # e parametro di fill  leal 4(%esp), %ecx # ecx = &p  movl %ecx, (%esp) # passaggio parametro  call fill  ... # altre operazioni  addl $12, %esp # dealloca spazio  ret |

## 3.3 Estensione a x86-64

Dopo avere familiarizzato con l'ISA IA32, mostreremo le differenze principali con la più moderna ISA a 64 bit dei processori x86. La prima semplice differenza è che per compilare a 64 bit su un sistema a 64 bit **non** bisogna inserire **-m32** come flag di gcc durante la compilazione. Vediamo ora altre differenze rilevanti.

### 3.3.1 Registri

Una delle differenze più visibili è la presenza di 16 registri general-purpose a 64 bit, rispetto agli 8 a 32 bit dell'IA32: A, B, C, D, DI, SI, SP, BP, R8, R9, R10, R11, R12, R13, R14, R15. Diversamente dall'ISA IA32, è possibile accedere agli 8 bit meno significativi dei registri DI e SI, denotati con DIL e SIL, rispettivamente. I registri general-purpose a 64 bit RAX, RBX, RCX, RDX, RDI, RSI, RSP, RBP estendono di ulteriori 32 bit i corrispondenti a 32 bit. Ogni scrittura dei 32 bit meno significativi di un registro a 64 bit azzera automaticamente i 32 bit più significativi. I registri R8-R15 possono essere acceduti nelle loro sottoparti meno significative usando i suffissi b (byte), w (word) e d (double word): es. R8 (registro a 64 bit), R8D (32 bit meno significativi), R8W (16 bit meno significativi), R8B (8 bit meno significativi).

### 3.3.2 Passaggio dei parametri alle funzioni

Diversamente dalle convenzioni dell'ABI System V 386 su IA32 dove tutti i parametri sono passati a una funzione in stack, in x86-64 i primi 6 parametri interi o puntatore sono passati rispettivamente nei registri RDI, RSI, RDX, RCX, R8, R9, e gli altri dopo il sesto in stack.

**Esempio**. Vediamo come effettuare una funzione swap che realizza lo scambio di due interi:

| .globl swap  swap:  movl (%rdi), %eax  movl (%rsi), %ecx  movl %eax, (%rsi)  movl %ecx, (%rdi)  ret |
| --- |

Si confronti la semplicità di questa realizzazione, dovuta al maggiore numero di registri che consente di usarli nel passaggio dei parametri, rispetto alla corrispondente IA32:

| .globl swap  swap:  pushl %ebx  movl 8(%esp), %eax  movl 12(%esp), %ecx  movl (%eax), %edx  movl (%ecx), %ebx  movl %edx, (%ecx)  movl %ebx, (%eax)  popl %ebx  ret |
| --- |

| #include <stdio.h>  void swap(int\*,int\*);  int main() {  int a = 10, b = 20;  swap(&a, &b);  printf("a=%d, b=%d\n", a, b);  return 0;  } |
| --- |

### 

### 3.3.3 Registri caller-save e callee-save

Nel [Paragrafo 3.2.3.4](#_giatdlobxgn8) abbiamo discusso le convenzioni dell'ABI System V 386 su IA32 che riguardano la responsabilità del salvataggio dei registri durante una chiamata a funzione. Le convenzioni dell'ABI System V su AMD64 sono leggermente diverse. In particolare, gli unici registri callee-save sono B, SP, BP e R12-R15. Gli altri sono tutti caller-save.

### 3.3.4 Suffissi delle istruzioni a 64 bit e istruzioni MOVS e MOVZ

E' possibile usare il suffisso q per indicare che un'istruzione lavora con operandi a 64 bit: es. movq $10, %rax. Inoltre le istruzioni MOVS e MOVZ possono essere usate in un contesto a 64 bit usando i suffissi b, w, l: es. movsbq (%rax), %rax legge un byte dall'indirizzo rax, ne estende il segno fino a riempire i 64 bit, e scrive il risultato in rax.

## 3.4 Domande riepilogative

Le seguenti domande riepilogative sugli argomenti del capitolo ammettono come risposte **vero** o **falso**. Salvo diversamente esplicitato, le domande si riferiscono all'ISA IA32.

1. La programmazione dei sistemi di calcolo si basano su astrazioni che nascondono dettagli irrilevanti dei sistemi di calcolo per i programmatori. Un esempio è l'ISA?
2. L'acronimo ISA sta per International Standard for Architectures?
3. L'ISA x86-64 è retrocompatibile con la IA32?
4. Il suffisso w si riferisce a operazioni su dati di 32 bit?
5. L'istruzione movb $5, %ax è corretta?
6. L'istruzione movl (%eax), (%ecx) è corretta?
7. Per rappresentare una variabile C di tipo int è possibile usare il registro %ecx?
8. Per assegnare il valore 0xABADCAFE al registro D si può usare l'istruzione movl %edx, $0xABADCAFE?
9. Il registro %esp viene usato per puntare all'indirizzo più basso della stack? VERO: la cima della stack, che cresce verso il basso (o verso sinistra nell’immagine della memoria logica del processo) ha l’indirizzo più basso. Lì punta %esp.
10. Gli 8 bit meno significativi del registro SI possono essere acceduti in IA32 mediante l'operando %sil?
11. Il registro %eip punta all'istruzione seguente che deve essere eseguita?
12. Un operando immediato può essere usato come operando destinazione?
13. Un operando immediato viene immagazzinato insieme all’istruzione stessa?
14. L'istruzione movl 10, %ecx scrive il numero 10 nel registro C?
15. Un operando memoria può essere rappresentato come (%ecx)?
16. Nella rappresentazione big-endian, il byte più significativo viene memorizzato per ultimo?
17. L'istruzione movzbl %al, %eax converte un unsigned char a un unsigned?
18. L'istruzione movsbl %al, %eax converte un char a un int?
19. Dato %ecx = 0x12341234, dopo l'esecuzione dell'istruzione movzbw %cl, %cx si ha %ecx = 0x00000034? FALSO: %cl = 4 = 0100 che viene propagato con 0 in %cx = 1 2 3 4 = 0001 0010 0011 0100 ottenendo %cx = 0000 0000 0000 0100 quindi in totale %ecx = 12340004
20. Dato %ecx = 0x12341234, dopo l'esecuzione dell'istruzione movsbw %cl, %cx si ha %ecx = 0x12340034? FALSO: %cl = 4 = 0100 che viene propagato con 0 in %cx = 1 2 3 4 = 0001 0010 0011 0100 ottenendo %cx = 0000 0000 0000 0100 quindi in totale %ecx = 12340004
21. Dato %ecx = 0x12348234, dopo l'esecuzione dell'istruzione movswl %cx, %ecx si ha %ecx = 0xFFFF8234? VERO: %cx = 8234 = 1000 0010 0011 0100 che viene propagato con 1 in %ecx ottenendo %ecx = 1111 1111 1111 1111 1000 0010 0011 0100 = FFFF8234
22. E' possibile mettere in cima alla stack un valore a 32 bit mediante l'istruzione popl?
23. Dato %ecx = 0xF0F0F0F0, dopo l'esecuzione notl %ecx si ha %ecx = 0x0F0F0F0F?
24. L'istruzione IDIV S calcola il quoziente e il resto della divisione del valore ottenuto concatenando il valore dei registri A e D (dividendo) per S (divisore), mettendo in A il resto e in D il quoziente?
25. L'istruzione leal (%eax), %ecx è equivalente all'istruzione movl %eax, %ecx?
26. L'istruzione leal -3(%eax,%ecx,4),%edx scrive in %edx il valore %eax+%ecx\*4-3?
27. Dato %eax=0xCAFEBABE, dopo l'istruzione sarl $4 si ha %eax=0xFCAFEBAB? VERO: %eax = 1100 1010 1111 1110 1011 1010 1011 1110 -> 1111 1100 1010 1111 1110 1011 1010 1011 cioè %eax = 0xFCAFEBAB
28. Dato %eax=0xCAFEBABE, dopo l'istruzione sall $4 si ha %eax=0xAFEBABEF? VERO: %eax = 1100 1010 1111 1110 1011 1010 1011 1110 -> 1010 1111 1110 1011 1010 1011 1110 0000 cioè %eax = 0xAFEBABE0
29. Dato %eax=0x00000001, dopo l'istruzione shll %eax si ha %eax=00000002? VERO: 0000 0000 0000 0000 0000 0000 0000 0001 -> 0000 0000 0000 0000 0000 0000 0000 0010 che è 0x00000002
30. Le istruzioni sal e shl sono equivalenti? VERO, hanno anche gli stessi opcode
31. Nel codice seguente, se %ecx=-2 allora l'istruzione je L salta all'indirizzo L?

| addl $2, %ecx  je L |
| --- |

1. L'istruzione cmpl %eax, $1 è valida?
2. L'istruzione call aumenta di 4 byte il valore di %esp?
3. L'istruzione test S, D è equivalente ad and S, D, tranne che test non modifica D?
4. Nell'istruzione DIV l'operando divisore può essere un immediato?
5. Nella tabella seguente, il codice IA32 è una corretta implementazione del codice C, assumendo che la variabile int a sia nel registro %eax?

| if (a>0) a++;  else a--;  ... | testl %eax, %eax  jle E  incl %eax  jmp F  E: decl %eax  F: ... |
| --- | --- |

1. Il seguente frammento di codice termina con un errore di stack overflow?

| L: incl %eax  jmp L |
| --- |

1. Le funzioni restituiscono sempre il proprio valore di ritorno, se a 16 bit, nel registro %ax?
2. I parametri passati a una funzione sono sempre a indirizzi inferiori a quelli dell'indirizzo di ritorno della funzione stessa?

FALSO: nella stack abbiamo prima i parametri passati alla funzione chiamata e poi RET della funzione chiamata. Questo perché prima mettiamo i parametri nella stack e poi chiamiamo call <funzione>, che mette RET. Quindi RET viene dopo, e poiché %esp punta a indirizzi più bassi (la stack cresce verso il basso, o verso sinistra se pensiamo al disegno della memoria logica del processo) si ha che RET ha un indirizzo più basso dei parametri.

# 4 Su quali librerie standard può contare un programmatore C?

Una **libreria** è una collezione coerente di funzioni progettate per un determinato scopo. Virtualmente qualsiasi programma utilizza funzioni di libreria, se non altro per procurarsi l'input e produrre l'output. In questo capitolo tratteremo alcune delle **principali funzioni della libreria standard** C, la libc, che ricade nello standard [POSIX](https://pubs.opengroup.org/onlinepubs/9699919799/). Naturalmente, un programmatore potrà utilizzare molte altre librerie progettate per compiti specifici che spesso non fanno parte delle distribuzioni standard dei sistemi operativi e che non tratteremo in questo capitolo.

In Figura 1.2 si vede come un programma C può invocare direttamente funzioni di libreria standard del linguaggio C, così come altre funzioni POSIX, che interfacciano il programma con il kernel del sistema operativo. Si noti che alcune funzioni della libreria standard C hanno bisogno esse stesse di richiamare funzioni POSIX, come ad esempio la funzione printf che necessita di produrre output testuale, tipicamente sul terminale. Altre invece, come la strlen, che calcola la lunghezza di una stringa, non hanno bisogno di usare altre librerie.

## 4.1 Accesso alla documentazione: man

Il comando man consente di accedere alla descrizione delle funzioni di libreria documentate nel sistema[[18]](#footnote-17), fra cui tutte quelle della libc. Riportiamo a titolo esemplificativo la parte iniziale della documentazione della funzione strlen mediante il comando: man strlen.

| STRLEN(3) Linux Programmer's Manual STRLEN(3)  NAME  strlen - calculate the length of a string  SYNOPSIS  #include <string.h>  size\_t strlen(const char \*s);  DESCRIPTION  The strlen() function calculates the length of the string pointed to by  s, excluding the terminating null byte ('\0').  RETURN VALUE  The strlen() function returns the number of bytes in the string pointed  to by s.  ... |
| --- |

La documentazione è divisa in capitoli:

1. **comandi** (programmi invocabili da riga di comando)
2. chiamate a sistema
3. librerie

Se il termine usato nella ricerca con man ricade in più di un capitolo è necessario specificare in quale cercare. Ad esempio, se siamo interessati alla documentazione della funzione standard C printf, è necessario digitare man 3 printf sulla riga di comando. Nella libc è infatti trattata nel primo e nel terzo capitolo, con significati diversi. Se invece siamo interessati alla chiamata a sistema write, scriveremo man 2 write.

## 4.2 Gestione degli errori

In questa sezione, discutiamo come gestire gli errori in un programma C in un ambiente Unix-like come Linux o MacOS X.

### 4.2.1 Codice di terminazione di un programma

Un processo ha sempre un **valore di ritorno intero** che viene passato all'indietro al processo che l'ha generato. Questo valore serve per segnalare se l'esecuzione ha avuto successo oppure è terminata per via di un errore. Il valore viene detto **codice di terminazione del programma** e vale 0 se il programma è terminato con successo, e diverso da zero altrimenti. Il codice di ritorno viene emesso da un programma invocando exit, \_exit (o \_Exit), abort, oppure con il return dal main.

### 4.2.2 Installare un gestore per la terminazione di un programma

Per garantire che la terminazione prematura di un programma in caso di errore non lasci pendenti operazioni che potrebbero compromettere esecuzioni future, come ad esempio un file lasciato troncato a metà, è possibile **installare un handler** (gestore) che viene eseguito automaticamente prima di terminare il programma. Per installare un handler, è possibile usare la funzione atexit, che può essere applicata anche nel caso di terminazione con successo come nell'esempio che segue:

| **#include <stdlib.h>**  **int atexit(void (\*function)(void));** |
| --- |
| **Parametri**:   * function: puntatore a una funzione da eseguire in caso di terminazione con la funzione exit oppure con return dal main |
| **Risultato**:   * 0 in caso di successo, -1 in caso di errore |
| **Esempio**:  #include <stdio.h> // printf  #include <stdlib.h> // atexit  void handler() {  printf("uscita dal programma\n");  }  int main() {  atexit(handler);  return 0; // stampa: "uscita dal programma"  } |

E' possibile installare un numero arbitrario di handler, che verranno eseguiti nell'ordine opposto a quello con cui sono stati installati con atexit.

### 4.2.3 Errori non recuperabili

Si tratta di errori che **provocano la terminazione del programma** e denotano un bug. Per segnalare un errore non recuperabile è possibile usare la macro C assert, che effettua un test e, se non verificato, fa terminare il programma:

| **#include <assert.h>**  **assert(test);** |
| --- |
| **Parametri**:   * test: valore di verità. Se zero, provoca la terminazione del programma con un codice diverso da zero e invia sul canale [stderr](#_c7lof5iy3ukf) un messaggio di errore della forma "Assertion failed: (<test>), function main, file <nomefile.c>, line <numero-linea>" |
| **Risultato**:   * nessuno |

### 4.2.4 Errori recuperabili

Un errore che viene individuato nel corpo di una funzione è **recuperabile** se è possibile continuare con l'esecuzione del programma, eventualmente effettuando delle operazioni di pulizia (cleanup) prima di uscire dalla funzione stessa. Ad esempio, potremmo volere scrivere una funzione che carica il contenuto di un file in un buffer in memoria. In caso di errore, il programma non dovrebbe terminare. Ove possibile, tutti gli errori dovrebbero essere recuperabili. Tratteremo due convenzioni con cui una funzione segnala al chiamante situazioni di errore. Per segnalare un errore:

1. la libreria standard C usa NULL nel caso in cui la funzione restituisce un puntatore e generalmente un valore diverso da 0 se la funzione restituisce un intero, ma ci sono casi in cui restituisce 0 in caso di errore (es. [strtol](https://pubs.opengroup.org/onlinepubs/9699919799/functions/strtol.html))
2. le funzioni dello standard [POSIX](https://pubs.opengroup.org/onlinepubs/9699919799/), escluse le librerie standard C, restituiscono 0 in caso di successo, e un valore diverso da 0 in caso di errore, tipicamente -1.

In entrambi i casi, viene settata una variabile globale chiamata errno dichiarata in errno.h con un codice di errore che specifica la natura del problema. E' anche possibile usare la funzione [perror](https://pubs.opengroup.org/onlinepubs/9699919799/functions/perror.html) definita in stdio.h per stampare l'errore generato dalla chiamata immediatamente precedente.

Un possibile **pattern di gestione degli errori recuperabili** è il seguente:

| int mia\_funzione() {  int res, \*buffer = NULL;  ...  buffer = malloc(...);  if (buffer == NULL) goto cleanup; // errore nella chiamata  ...  res = chiamata(...);  if (res != 0) goto cleanup; // errore nella chiamata  ...  free(buffer);  return 0;  cleanup:  ...  if (buffer != NULL) free(buffer); // deallocazione buffer  return -1;  } |
| --- |

L'esempio mostra come impostare una funzione in modo che in caso di errore vengano liberate tutte le risorse allocate dalla funzione stessa. Si noti l'uso del goto, che normalmente va evitato perché rende meno leggibile e manutenibile il codice [Dijkstra 1968], ma che si presta bene al caso in questione.

## 4.3 Panoramica della libreria standard C

In questa sezione introduciamo alcune delle funzioni più comunemente usate nella scrittura di programmi C, rimandando il lettore all'uso del comando man per tutte le altre.

### 4.3.1 Stringhe: strlen, strcpy, strcat, strcmp, sprintf, sscanf, strtok, atoi

Calcolo della lunghezza di una stringa:

| **#include <string.h>**  **size\_t strlen(const char \*s);** |
| --- |
| **Parametri**:   * s: stringa in input |
| **Risultato**:   * numero di caratteri nella stringa (escluso il terminatore zero) |

Copia di una stringa:

| **#include <string.h>**  **char \*strcpy(char \*dest, const char \*src);** |
| --- |
| **Parametri**:   * dest: puntatore a buffer di output in cui copiare la stringa, che deve contenere spazio sufficiente per includere i caratteri della stringa puntata da src, più il terminatore '\0' * src: puntatore alla stringa da copiare |
| **Risultato**:   * puntatore dest |
| **Esempio:**  char dest[6]; // 5 char per la stringa + 1 per il terminatore  strcpy(dest, "hello"); // copia "hello" nel buffer dest  printf("%s\n", dest); // stampa: hello |

Concatenazione di stringhe:

| **#include <string.h>**  **char \*strcat(char \*dest, const char \*src);** |
| --- |
| **Parametri**:   * dest: puntatore a buffer di output a cui concatenare la stringa, che deve contenere spazio sufficiente per includere i caratteri della stringa in dest, più la stringa puntata da src, più il terminatore '\0' * src: puntatore alla stringa da concatenare |
| **Risultato**:   * puntatore dest |
| **Esempio:**  char dest[12] = "hello"; // 5 char per la prima stringa + 6 per la  // seconda + 1 per il terminatore  strcat(dest, " world"); // concatena "world" al buffer dest  printf("%s\n", dest); // stampa: hello world |

**Nota**: si faccia attenzione nell'uso delle funzioni strcpy e strcat, poiché potrebbero portare a un errore di **buffer overflow** se non c'è spazio sufficiente nella destinazione per ricevere i caratteri copiati. Per imporre un limite al numero di caratteri copiati, si suggerisce di usare le varianti strncpy e strncat (si veda il man per maggiori dettagli).

Confronto lessicografico tra stringhe:

| **#include <string.h>**  **int strcmp(const char \*s1, const char \*s2);** |
| --- |
| **Parametri**:   * s1: puntatore alla prima stringa * s2: puntatore alla seconda stringa |
| **Risultato**:   * intero negativo se s1 precede s2 nell'ordinamento lessicografico, 0 se le stringhe sono uguali, e un intero positivo se s2 precede s1 nell'ordinamento lessicografico |
| **Esempio:**  strcmp("abacus", "root") // restituisce intero negativo  strcmp("root", "root") // restituisce zero  strcmp("root", "abacus") // restituisce intero positivo |

Formattazione di stringhe:

| **#include <stdio.h>**  **int sprintf(char \*str, const char \*format, ...);** |
| --- |
| **Parametri**:   * str: puntatore alla stringa destinazione * format: puntatore alla stringa di formato (si veda man 3 printf) * **...**: argomenti della sprintf (si veda man 3 printf) |
| **Risultato**:   * in caso di successo, numero di caratteri scritti nella destinazione, escluso il terminatore * in caso di errore, un intero negativo |
| **Esempio:**  char buffer[9];  sprintf(buffer, "(%d,%.1f)", 10, 3.14);  printf("%s\n", buffer); // stampa (10,3.1) |

In modo simmetrico, è possibile smontare una stringa nelle sue diverse parti usando la funzione sscanf come segue:

| **#include <stdio.h>**  **int sscanf(char \*str, const char \*format, ...);** |
| --- |
| **Parametri**:   * str: puntatore alla stringa sorgente * format: puntatore alla stringa di formato (si veda man scanf) * **...**: argomenti della sscanf (si veda man scanf) |
| **Risultato**:   * in caso di successo, numero di argomenti letti dalla stringa, oppure EOF se la stringa non contiene sufficienti dati per leggere gli argomenti * in caso di errore, un intero negativo |
| **Esempio:**  #include <stdio.h>  int main() {  char\* str = "a 13 2.7";  int i;  float d;  int res = sscanf(str, "a %d %f", &i, &d);  printf("%d: %d %f\n", res, i, d); // stampa 2: 13 2.7  return 0;  } |

Con la strtok è invece possibile tokenizzazione stringhe, ovvero suddividerle in parole divise da un elenco di possibili separatori:

| **#include <string.h>**  **char \*strtok(char \*str, const char \*delim);** |
| --- |
| **Parametri**:   * str: stringa iniziale da tokenizzare (NULL per le chiamate successive alla prima) * delim: stringa i cui caratteri rappresentano possibili delimitatori fra parole |
| **Risultato**:   * puntatore al token identificato, oppure NULL se i token sono esauriti |
| **Esempio:** (il programma stampa su righe diverse hello e world)  char str[] = "hello, world!";  char \*token = strtok(str, "! ,");  while (token != NULL) {  printf("%s\n", token);  token = strtok(NULL, "! ,");  } |

**Attenzione**: la funzione **modifica il buffer di input** puntato da str, sostituendo a ogni occorrenza di un separatore il carattere terminatore '\0'.

Concludiamo il paragrafo con un esempio di funzione, atoi, che converte una stringa contenente un numero a un tipo numerico.

| **#include <stdlib.h>**  **int atoi(const char \*nptr);** |
| --- |
| **Parametri**:   * nptr: stringa da convertire a numero |
| **Risultato**:   * intero rappresentato dal prefisso della stringa che rappresenta un numero |
| **Esempio:**  int num = atoi("971esempio");  printf("%d\n", num); // stampa 971  num = atoi("esempio");  printf("%d\n", num); // stampa 0 |

Vi sono le ulteriori varianti atol e atoll, che convertono rispettivamente a long e long long, e strtol, che gestisce situazioni di errore per cui rimandiamo alla documentazione man. Come variante di atoi è possibile usare la funzione sscanf.

### 4.3.2 Memoria: memcpy, memcmp, memset

La funzione memcpy copia di un blocco di byte:

| **#include <string.h>**  **void \*memcpy(void \*dest, const void \*src, size\_t n);** |
| --- |
| **Parametri**:   * dest: puntatore a buffer di output in cui copiare il blocco puntato da src, che deve contenere spazio sufficiente per includere i byte puntati da src * src: puntatore alla buffer da cui copiare * n: numero di byte da copiare |
| **Risultato**:   * puntatore dest |
| **Esempio:**  char dest[6]; // 6 char per la stringa incluso terminatore  memcpy(dest, "hello", 6); // copia "hello" nel buffer dest  printf("%s\n", dest); // stampa: hello |

La funzione memcmp confronta due blocchi di byte:

| **#include <string.h>**  **int memcmp(const void \*s1, const void \*s2, size\_t n);** |
| --- |
| **Parametri**:   * s1: puntatore a buffer * s2: puntatore a buffer * n: numero di byte da confrontare |
| **Risultato**:   * restituisce un intero positivo se a>b e negativo se a<b, dove a e b sono la prima coppia di byte in s1 e s2 che differiscono fra i primi n byte. Restituisce zero se le prime n coppie di byte nei buffer sono uguali. |
| **Esempio:**  int r = memcmp("hell", "hello", 4);  printf("%d\n", r); // stampa 0  r = memcmp("ant", "Ant", 3);  printf("%d\n", r); // stampa intero > 0  r = memcmp("Ant", "ant", 3);  printf("%d\n", r); // stampa intero < 0 |

La funzione memset setta infine tutti i byte di un buffer ad uno specifico valore:

| **#include <string.h>**  **void \*memset(void \*s, int c, size\_t n);** |
| --- |
| **Parametri**:   * s: puntatore a buffer * c: intero da scrivere in ciascuna cella del buffer (vengono presi gli 8 bit meno significativi di c) * n: dimensione del buffer in byte |
| **Risultato**:   * puntatore s |
| **Esempio:**  char dest[6];  memset(dest, '\*', 5); // setta i primi 5 byte di dest a '\*'  dest[5] = '\0'; // aggiunge terminatore  printf("%s\n", dest); // stampa \*\*\*\*\* |

### 4.3.3 File e stream

Le chiamate di libreria che tratteremo in questo paragrafo riguardano sia la scrittura che la lettura di dati da terminale e da file. Useremo il termine stream per denotare un generico canale di I/O.

#### 4.3.3.1 Canali standard di input/output: stdin, stdout, stderr

Di default, quando un processo viene avviato in un sistema Unix-like, tre canali di comunicazione (anche detti [standard stream](https://en.wikipedia.org/wiki/Standard_streams)) vengono inizialmente creati. I tre canali sono comunemente denominati:

* standard input (stdin): viene mappato di default all’input generato dalla tastiera. Il canale viene automaticamente utilizzato da diverse funzioni C: ad esempio gets(), getc(), e getchar() leggono dati provenienti da questo canale.
* standard output (stdout): 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): 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. Il 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 mescolato, sul terminale. Nel paragrafo seguente vedremo come creare stream (canali) collegati a file.

#### 4.3.3.2 Apertura e chiusura di file: fopen e fclose

Per poter accedere a un file è necessario innanzitutto aprirlo con fopen:

| **#include <stdio.h>**  **FILE \*fopen(const char \*pathname, const char \*mode);** |
| --- |
| **Parametri**:   * pathname: percorso e nome del file da aprire * mode: modalità di apertura del file (si veda man 3 fopen) |
| **Risultato**:   * in caso di successo, puntatore a oggetto FILE che rappresenta il file (canale) aperto * in caso di errore, NULL |
| **Esempio:** creazione di un file di testo nella directory corrente  #include <stdio.h>  #include <assert.h>  int main() {  FILE\* f = fopen("miofile.txt", "w");  assert(f != NULL);  fprintf(f, "hello world");  fclose(f);  return 0;  } |

Per suggellare la fine delle attività su un file è necessario chiuderlo con fclose.

| **#include <stdio.h>**  **int fclose(FILE \*stream);** |
| --- |
| **Parametri**:   * stream: percorso e nome del file da chiudere |
| **Risultato**:   * in caso di successo restituisce 0 * in caso di errore restituisce la costante EOF e imposta errno con il codice di errore |
| **Esempio:** si veda l'esempio mostrato per la funzione fopen |

#### 4.3.3.3 Lettura e scrittura di file di testo: printf, fprintf, scanf, fscanf, fgets

Una delle chiamate più comuni per inviare testo su un canale di output è la printf:

| **#include <stdio.h>**  **int printf(const char \*format, ...);** |
| --- |
| **Parametri**:   * format: stringa di formato (si veda man 3 printf) * ...: argomenti relativi alla stringa di formato (si veda man 3 printf) |
| **Risultato**:   * in caso di successo, numero di caratteri scritti nella destinazione, escluso il terminatore * in caso di errore, un intero negativo |
| **Esempio:**  int main() {  // stampa: hello hello hello world  printf("%1$s %1$s %s %s\n", "hello", "world");  return 0;  } |

Più articolata è invece la funzione fprintf, che consente anche di scrivere su un file aperto:

| **#include <stdio.h>**  **int fprintf(FILE \*stream, const char \*format, ...);** |
| --- |
| **Parametri**:   * stream: riferimento al flusso di output, che può essere un file aperto, oppure le costanti stderr o stdout, che denotano i canali di output standard per la messaggistica di errore e quella regolare di un programma, rispettivamente. Si noti che fprintf(stdout, ...) è equivalente a printf(...) * format: stringa di formato (si veda man 3 printf) * ...: argomenti relativi alla stringa di formato (si veda man 3 printf) |
| **Risultato**:   * in caso di successo, numero di caratteri scritti nella destinazione, escluso il terminatore * in caso di errore, un intero negativo |
| **Esempio:** si veda l'esempio mostrato per la funzione fopen |

Si noti che printf(...) è equivalente a fprintf(stdout, ...).

Per leggere testo da uno stream (es. terminale) o file di input si può usare scanf o fscanf:

| **#include <stdio.h>**  **int scanf(const char \*format, ...);**  **int fscanf(FILE \*stream, const char \*format, ...);** |
| --- |
| **Parametri**:   * stream: flusso di input, per fscanf * format: stringa di formato (si veda man scanf) * ...: argomenti relativi alla stringa di formato (si veda man scanf) |
| **Risultato**:   * in caso di successo, numero di argomenti letti dal file * in caso di errore, un intero negativo |
| **Esempio:**  #include <stdio.h>  int main() {  FILE\* f = fopen("prova.txt", "r");  for (;;) {  int x = 0, y = 0, res;  res = fscanf(f, "a %d %d\n", &x, &y);  if (res < 2)  res = fscanf(f, "b %d %d\n", &x, &y);  if (res < 2)  res = fscanf(f, "c %d %d\n", &x, &y);  printf("%d: %d %d\n", res, x, y);  if (res == 0 || res == EOF) break;  }  fclose(f);  return 0;  } |
| prova.txt:  c 1 2  a 3 4  b 6 0  a 7 8  d 9 10  a 1 3 |
| output del programma (si noti che si ferma su d 9 10 poiché il caso d non è trattato):  2: 1 2  2: 3 4  2: 6 0  2: 7 8  0: 0 0 |

Si noti che scanf(...) è equivalente a fscanf(stdin, ...).

Concludiamo la trattazione dei file di testo con la fgets, che consente di leggere una intera riga di un file:

| **#include <stdio.h>**  **char\* fgets(char \*str, int size, FILE \*stream);** |
| --- |
| **Parametri**:   * str: buffer in cui scrivere i caratteri * size: dimensione del buffer puntato da str * stream: flusso di input |
| **Risultato**:   * in caso di successo, restituisce str in cui sono stati copiati tutti i caratteri di una riga, compreso il carattere di ritorno a capo. Il risultato è una stringa C terminata dal carattere '\0'. E' necessario quindi che il buffer str sia di dimensioni almeno size+1 per fare spazio al terminatore di stringa. * in caso di errore, restituisce NULL |
| **Esempio:**  #include <stdio.h>  #include <assert.h>  #include <stdlib.h>  #define N 127  int main() {  char \*line = malloc((N+1)\*sizeof(char));  char \*res = fgets(line, N, stdin);  assert(res != NULL);  printf("%s", res); // il ritorno a capo è già in res  free(line);  return 0;  } |

#### 4.3.3.4 Lettura e scrittura di file binari: fwrite e fread

Per scrivere byte verso un flusso di input è possibile usare la funzione fwrite:

| **#include <stdio.h>**  **size\_t fwrite(const void \*ptr, size\_t size, size\_t nitems, FILE \*stream);** |
| --- |
| **Parametri**:   * ptr: puntatore a buffer da cui leggere i byte * size: dimensione in byte degli elementi di ptr * nitems: numero di elementi da scrivere sul flusso * stream: flusso su cui scrivere |
| **Risultato**:   * numero di elementi scritti sul file (fwrite fa avanzare la posizione corrente nel file del numero di elementi scritti) |
| **Esempio:**  #include <stdio.h>  #include <assert.h>  #define N 5  int main() {  int buffer[N] = { 7, 12, 5, 8, 23 };  FILE \*f = fopen("numeri.dat", "w");  assert(f != NULL);  int numelem = fwrite(buffer, sizeof(int), N, f);  fclose(f);  printf("%d interi scritti\n: ", numelem);  return 0;  } |

Per leggere byte da un flusso di input è possibile usare la funzione fread:

| **#include <stdio.h>**  **size\_t fread(void \*ptr, size\_t size, size\_t nitems, FILE \*stream);** |
| --- |
| **Parametri**:   * ptr: puntatore a buffer in cui scrivere i byte * size: dimensione in byte degli elementi di ptr * nitems: numero di elementi da leggere dal flusso * stream: flusso da cui leggere |
| **Risultato**:   * numero di elementi letti dal file (fread fa avanzare la posizione corrente nel file del numero di elementi letti) |
| **Esempio:**  #include <stdio.h>  #include <assert.h>  #include <stdlib.h>  #define N 1024 // massimo numero di interi da leggere  int main() {  int \*buffer = malloc(N\*sizeof(int));  assert(buffer != NULL);  FILE \*f = fopen("numeri.dat", "r"); // usare esempio fwrite  assert(f != NULL);  int i, numelem = fread(buffer, sizeof(int), N, f);  fclose(f);  printf("%d interi letti: ", numelem);  for (i=0; i<numelem; ++i) printf("%d ", buffer[i]);  printf("\n");  free(buffer);  return 0;  } |

### 4.3.4 Ricerca e ordinamento: bsearch, qsort

La libc contiene supporto per ricerca binaria e per ordinamento. La funzione di ricerca binaria assume che l'array sia ordinato:

| **#include <stdlib.h>**  **void \*bsearch(const void \*key, const void \*base, size\_t nel,**  **size\_t width, int (\*compar)(const void \*, const void \*));** |
| --- |
| **Parametri**:   * key: puntatore a buffer contenente la chiave da cercare nell'array * base: array di elementi ordinato per chiave di ricerca * nel: numero di elementi nell'array * width: dimensione in byte di un elemento dell'array * compar: funzione comparatore che restituisce un intero negativo se il primo elemento precede il secondo, zero se gli elementi sono uguali, e un intero positivo altrimenti |
| **Risultato**:   * puntatore all'elemento trovato in base, oppure NULL se l'elemento non è stato trovato. Se vi è più di una occorrenza della chiave, ne viene restituita una a caso |
| **Esempio:**  int comparator(const void \*x, const void \*y) {  int xt = \*(int\*)x;  int yt = \*(int\*)y;  return xt-yt;  }  int main() {  int v[] = { 2, 4, 5, 7, 9 }, key = 7;  int \*res = bsearch(&key, v, 5, sizeof(int), comparator);  printf("%d\n", \*res); // stampa 7  return 0;  } |

La funzione di ordinamento standard è descritta di seguito:

| **#include <stdlib.h>**  **void qsort(void \*base, size\_t nel, size\_t width,**  **int (\*compar)(const void \*, const void \*));** |
| --- |
| **Parametri**:   * base: array di elementi da ordinare * nel: numero di elementi nell'array * width: dimensione in byte di un elemento dell'array * compar: funzione comparatore che restituisce un intero negativo se il primo elemento precede il secondo, zero se gli elementi sono uguali, e un intero positivo altrimenti |
| **Risultato**:   * nessuno. Come side-effect la funzione ordina l'array base in base al comparatore fornito |
| **Esempio:**  int comparator(const void \*x, const void \*y) {  int xt = \*(int\*)x, yt = \*(int\*)y;  return xt-yt;  }  int main() {  int v[] = { 7, 4, 5, 3, 2 };  qsort(v, 5, sizeof(int), comparator);  // stampa 2 3 4 5 7  printf("%d %d %d %d %d\n", v[0], v[1], v[2], v[3], v[4]);  return 0;  } |

### 4.3.5 Funzioni matematiche

Per usare le funzioni matematiche è necessario aggiungere lo switch -lm alla riga di comando per includere la libreria matematica. Per consultare quali librerie matematiche e costanti notevoli sono disponibili si rimanda al comando man math.h. Riportiamo a titolo di esempio la funzione per il calcolo della radice quadrata:

| **#include <math.h>**  **double sqrt(double x);** |
| --- |
| **Parametri**:   * x: valore di input |
| **Risultato**:   * radice quadrata di x se x è non negativo, zero se x è zero, e la costante NaN altrimenti, impostando errno alla costante EDOM |
| **Esempio:**  printf("%f\n", sqrt(2)); // stampa 1.414214 |

## 4.4 Tracciamento delle system call: strace

Il comando strace è progettato per analizzare il flusso di chiamate al sistema operativo effettuate da un programma. Si usa invocando strace nome-programma.

Consideriamo il seguente programma compilato con gcc hello.c -o hello -O1:

| #include <stdio.h>  int main() {  printf("hello world\n");  return 0;  } |
| --- |

L'analisi dell'esecuzione del programma hello sotto il controllo di strace fornisce il seguente output, di cui riportiamo solo la parte finale:

| $ strace ./hello > /dev/null  ...  arch\_prctl(ARCH\_SET\_FS, 0x7fdf1094f540) = 0  mprotect(0x7fdf10944000, 12288, PROT\_READ) = 0  mprotect(0x55bab5c48000, 4096, PROT\_READ) = 0  mprotect(0x7fdf109a6000, 4096, PROT\_READ) = 0  munmap(0x7fdf10950000, 178959) = 0  fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0), ...}) = 0  brk(NULL) = 0x55bab6e53000  brk(0x55bab6e74000) = 0x55bab6e74000  **write(1, "hello world\n", 12) = 12**  **exit\_group(0) = ?**  +++ exited with 0 +++  $ |
| --- |

Si noti che strace emette il proprio output sul canale stderr (si veda [Appendice B](#_8fyy8diz8l9k)), mentre il programma hello usa stdout. Per visualizzare solo l'output di strace basta redirigere stdout sul dispositivo virtuale /dev/null che funge da "pozzo senza uscita" usato per scartare dei dati prodotti da un programma, come visto nel programma di prova sopra.

La parte iniziale dell'output di strace include numerose chiamate a sistema (di cui ne riportiamo solo alcune) che avvengono prima dell'esecuzione del main per impostare l'ambiente di esecuzione. La printf finisce per invocare la system call write sul canale 1, che rappresenta il canale di standard output, normalmente indirizzato verso il terminale. Questo rappresenta un esempio di funzione libc che invoca a sua volta system call come illustrato in Figura 1.2. Il return 0 porta alla terminazione del programma che si conclude con una chiamata a sistema exit\_group(0).

## 4.5 Tracciamento delle funzioni di libreria: ltrace

In questo paragrafo trattiamo il comando ltrace, che intercetta le chiamate alle librerie dinamiche effettuate da un programma.

Consideriamo il seguente programma compilato con gcc mytest.c -o mytest -lm:

| #include <math.h>  #include <stdio.h>  double f(double x){  return sqrt(x);  }  int main() {  double y = f(2.0);  printf("%f\n", y);  return 0;  } |
| --- |

L'analisi dell'esecuzione del programma mytest sotto il controllo di ltrace fornisce il seguente output:

| $ ltrace ./mytest > /dev/null  **sqrt**(1, 0x7ffca1dee0e8, 0x7ffca1dee0f8, 0x7fa8fd1d6578)  = 0x4000000000000000  **printf**("%f\n", 1.414214) = 9  +++ exited (status 0) +++  $ |
| --- |

Si osservi che ltrace intercetta le due chiamate a libreria sqrt e printf.

# 

# 5 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 possibile[[19]](#footnote-18).

**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 affronteremo 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.

## 

Figura 5.1: legge di Moore e andamento storico del potere computazionale dei calcolatori

## 5.1 Quanto è importante ottimizzare le prestazioni?

I programmatori professionisti sono abituati a considerare numerose proprietà del software ancor prima di pensare alle prestazioni. **C**[**orrettezza**](http://it.wikipedia.org/wiki/Correttezza), [**affidabilità**](http://it.wikipedia.org/wiki/Affidabilit%C3%A0)**,** [**robustezza**](http://it.wikipedia.org/wiki/Robustezza_(informatica))**,** [**usabilità**](http://it.wikipedia.org/wiki/Usabilit%C3%A0)**,** [**verificabilità**](http://it.wikipedia.org/wiki/Verificabilit%C3%A0)**,** [**manutenibilità**](http://it.wikipedia.org/wiki/Manutenibilit%C3%A0)**,** [**riparabilità**](http://it.wikipedia.org/w/index.php?title=Riparabilit%C3%A0&action=edit&redlink=1)**,** [**evolvibilità**](http://it.wikipedia.org/w/index.php?title=Evolvibilit%C3%A0&action=edit&redlink=1)**,** [**riusabilità**](http://it.wikipedia.org/wiki/Riusabilit%C3%A0)**,** [**portabilità**](http://it.wikipedia.org/wiki/Portabilit%C3%A0)**,** [**leggibilità**](http://it.wikipedia.org/wiki/Leggibilit%C3%A0)e [**modularità**](http://it.wikipedia.org/wiki/Modularit%C3%A0)sono 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. Gordon Moore già negli anni '70 aveva infatti predetto che il numero di transistor sarebbe raddoppiato ogni 18 mesi, correggendo poi la stima a un raddoppio ogni 24 mesi. La previsione è stata talmente accurata che è universalmente nota come "legge di Moore" (Figura 5.1).

Come si vede nella Figura 5.1, 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.

## 5.2 Quali parti di un programma ottimizzare e cosa 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. Stimareil **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**.

### 5.2.1 Metriche di prestazioni

Nell'ottimizzazione delle prestazioni vi sono svariati obiettivi possibili che dipendono dalla particolare applicazione che si intende costruire. Ad esempio, potremmo essere interessati a una bassa **latenza** delle operazioni svolte dall'applicazione, vale a dire la durata dall'inizio alla fine delle singole operazioni. Oppure, potremmo volere massimizzare il **throughput**, cioè il numero di operazioni completate nell'unità di tempo, indipendentemente dalla durata di singole operazioni che potrebbero essere più lente di altre. In alcune applicazioni che lavorano con grandi moli di dati la metrica di interesse potrebbe essere lo **spazio** occupato dai dati. Potrebbero esserci anche altre metriche di interesse, come il **consumo di energia elettrica**, in particolare se il software viene eseguito su applicazioni mobili.

### 5.2.2 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[[20]](#footnote-19)** | **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 𝜇sec | 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 |

Riportiamo sia una stima della latenza effettiva dell'operazione che una latenza riscalata che fornisce un'intuizione delle differenze relative di latenza tra operazioni diverse.

### 5.2.3 Speedup e legge di Amdahl

La legge di Amdahl[[21]](#footnote-20) 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 il tempo di esecuzione di un programma e sia il tempo di esecuzione dopo aver applicato un'ottimizzazione. Il rapporto 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 il programma ottimizzato è più veloce di quello originario.

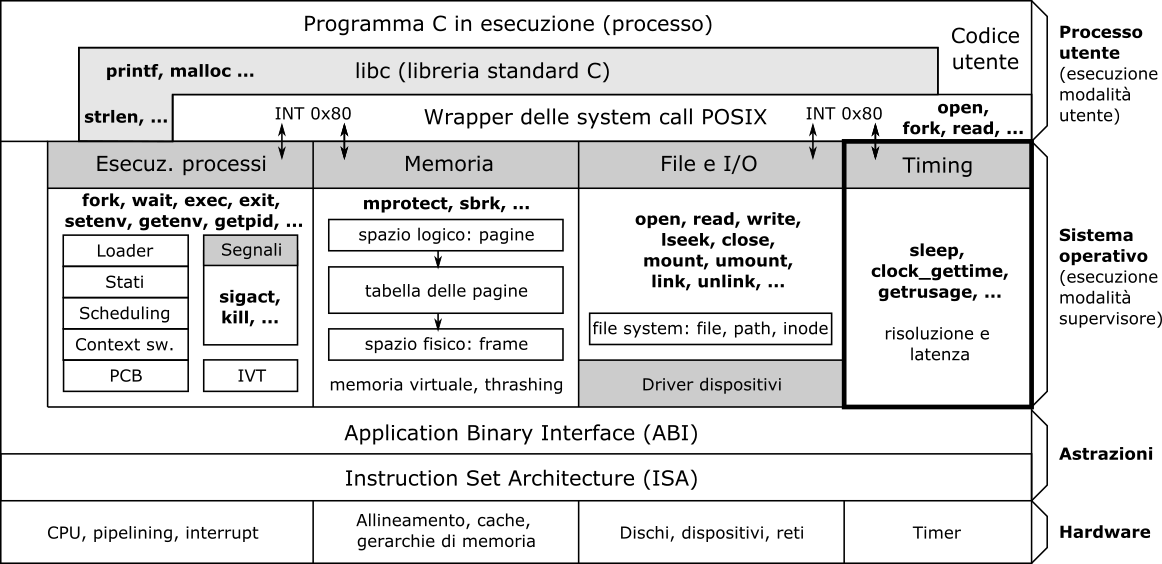
**Esempio 1.**

Supponiamo che un programma richieda secondi su determinati dati di input. Dopo aver ottimizzato il programma, il tempo di esecuzione sugli stessi dati di input diventa secondi. Lo speedup dovuto all'ottimizzazione è . Per indicare che il programma ottimizzato è 1.42 volte più veloce di quello non ottimizzato usiamo la notazione 1.42x.

| **Legge di Amdahl.** Supponiamo di dividere un programma in due parti e . Sia T il tempo totale speso dal programma, sia il tempo speso dal programma in e sia il tempo speso in . Supponiamo di ottimizzare in modo che sia volte più veloce. Lo speedup ottenuto è:  *Dimostrazione.* Si ha: , da cui si ottiene: |
| --- |

**Esempio 2.**

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 e . Applicando la legge di Amdahl otteniamo: .



**Figura 5.2.1.1**: system call di misurazione del tempo in Linux

### 5.2.4 Profilazione delle prestazioni

La profilazione delle prestazioni consiste nell'analizzare il tempo (o qualche altra metrica) 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.

#### 5.2.4.1 System call di misurazione del tempo

I sistemi POSIX offrono varie chiamate a sistema per la misurazione del tempo, come illustrato in Figura 5.2.1.1. 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). Le system call clock e getrusage, misurando il **tempo speso da un processo nell'uso della CPU**, non permettono di contabilizzare il tempo in cui un processo è rimasto in attesa di un evento senza usare la CPU. 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 al 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.

Un altro aspetto importante è se la funzione di misurazione calcola il tempo reale (come clock\_gettime) oppure quello contabilizzato su base processo (come getrusage). Nel caso di misurazione di tempo reale abbiamo alta precisione, ma il valore ottenuto potrebbe soffrire dall'eventuale presenza di altri processi che consumano tempo di CPU. Nel caso di tempo utente/sistema la misura contabilizza il tempo speso da un processo, ma potrebbe soffrire di minore precisione.

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](http://pubs.opengroup.org/onlinepubs/009695399/functions/clock_getres.html) 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;  } |
| --- |

Vediamo infine come utilizzare la system call getrusage:

| **#include <sys/resource.h>**  **int getrusage(int who, struct rusage \*r\_usage)** |
| --- |
| **Descrizione:** la funzione riporta varie informazioni sull'uso delle risorse da parte del processo chiamante. In questa scheda ci concentriamo sul tempo speso dal processo. |
| **Parametri**:   * int who: può essere RUSAGE\_SELF, che effettua le misurazioni sul processo chiamante, oppure RUSAGE\_CHILDREN, che considera invece i figli del processo chiamante; * struct rusage \*r\_usage: puntatore a un buffer che riceve il tempo misurato e altre informazioni.  | **struct rusage** | | --- | | **Attributi**:   * struct timeval ru\_utime: tempo totale speso dal processo in modalità utente * struct timeval ru\_stime: tempo totale speso dal processo in modalità sistema * (si veda la [documentazione online](http://pubs.opengroup.org/onlinepubs/009695399/functions/clock_getres.html) per altri attributi) |  | **struct timeval** | | --- | | **Attributi:**   * time\_t tv\_sec: tempo speso in secondi * suseconds\_t tv\_usec: tempo speso in microsecondi, da aggiungere a tv\_sec | |
| **Risultato**:   * In caso di successo, restituisce 0 * In caso di errore, restituisce -1 ed errno specifica i dettagli dell'errore |

**Esempio**: si consideri lo stesso esempio visto per clock\_gettime, sostituendo la funzione get\_real\_time\_msec con get\_user\_time\_usec:

| #include <sys/resource.h> // getrusage  double get\_user\_time\_msec() {  struct rusage ru;  getrusage(RUSAGE\_SELF, &ru);  return ru.ru\_utime.tv\_sec\*1E03 + ru.ru\_utime.tv\_usec\*1E-03;  } |
| --- |

#### 5.2.4.2 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. |

## 5.3 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.

### 5.3.1 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 Mint[[22]](#footnote-21). Si noti che versioni più recenti o comunque diverse di gcc potrebbero dare risultati diversi. 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-4.7.2/gcc/Optimize-Options.html#Optimize-Options)[[23]](#footnote-22). 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.

### 5.3.2 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(n2) 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**.

#### 5.3.2.1 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**.

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

| **C originale: test-cf.c** | **Constant folding (manualmente)** |
| --- | --- |
| int f() {  return 8+(14/2)\*3;  } | int f() {  return 29;  } |

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

| **gcc -O1 -S -fomit-frame-pointer test-cf.c** |
| --- |
| f: movl $29, %eax  ret |

#### 5.3.2.2 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)** |
| --- | --- | --- |
| int x;  int f() {  x = 8;  return x-2;  } | int x;  int f() {  x = 8;  return 8-2;  } | int x;  int f() {  x = 8;  return 6;  } |

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.

| **gcc -O1 -S -fomit-frame-pointer -m32 test-cp.c** |
| --- |
| f: movl $8, x  movl $6, %eax  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.

#### 5.3.2.3 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)** |
| --- | --- |
| int expr(int x, int y) {  return (x + y)\*(x + y);  } | int expr(int x, int y) {  int z = x + y;  return z \* z;  } |

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.

#### 5.3.2.5 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 la 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)** |
| --- | --- |
| int f(int x, int n) {  int s = 1;  for (; n>0; n--) s \*= 7+x;  return s;  } | int f(int x, int n) {  int s = 1, **z = 7+x**;  for (; n>0; n--) s \*= **z**;  return s;  } |

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 -O1** |
| --- |
| 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 |

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.)** |
| --- | --- |
| int has\_space(const char\* s) {  int i;  for (i=0; i<strlen(s); i++)  if (s[i]==' ') return 1;  return 0;  } | int has\_space(const char\* s) {  int i, **n = strlen(s)**;  for (i=0; i<**n**; i++)  if (s[i]==' ') return 1;  return 0;  } |

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 rimanga sempre la stessa. Si noti che il costo della funzione has\_space è 𝛩(n2), dove n è la lunghezza della stringa s passata in ingresso! Questo è a tutti gli effetti un **performance bug[[24]](#footnote-23)** 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.

#### 5.3.2.6 Loop unrolling

Il loop unrolling (srotolamento di un ciclo) è un'ottimizzazione spesso effettuata direttamente dai compilatori. Lo scopo è quello di ridurre di una frazione costante k il numero di iterazioni di un ciclo, effettuando in una sola iterazione le operazioni che verrebbero effettuate su k iterazioni consecutive. Questo consente di abbassare il work riducendo il numero di salti e di test del ciclo. Illustriamo la tecnica con un esempio.

**Esempio**. Partiamo da una semplice implementazione della somma degli elementi di un array e riduciamo il numero di iterazioni di un fattore 3:

| **C originale: test-lu.c** | **Loop unrolling di fattore 3 (manualmente)** |
| --- | --- |
| int sum(int \*v, int n) {  int i, s = 0;  for (i=0; i<n; ++i) s += v[i];  return s; } | int sum(int \*v, int n) {  int i, s = 0;  for (i=0; i+2<n; i+=3) {  s += v[i];  s += v[i+1];  s += v[i+2];  }  for (; i<n; ++i) s += v[i];  return s; } |

Si noti che il secondo ciclo for nel loop unrolling è necessario se l'array ha una dimensione che non è un multiplo di 3.

#### 5.3.2.7 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 function 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.

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)** |
| --- | --- |
| 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);  } | void f(int\* v, int n) {  int i;  for (i=0; i<n; i++)  v[i] = **i\*i**;  } |

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 -O3** |
| --- |
| 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  ... |

### 5.3.3 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.

#### 5.3.3.1 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 alle variabili più frequentemente usate 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: test-ra.c** | **Senza alloc. registri (-O0)** | **Con alloc. registri (-O1)** |
| --- | --- | --- |
| void f(int n) {  while (n--) g();  } | 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 | f: pushl %ebx  movl 8(%esp), %ebx  testl %ebx, %ebx  je L3  L5: call g  subl $1, %ebx  jne L5  L3: popl %ebx  ret |

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[[25]](#footnote-24) e nel ciclo viene tenuta in quel registro, velocizzando l'accesso.

#### 5.3.3.2 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 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 >> 3 |
| x \* 64 | x << 6 |
| x \* 2 | x << 1 oppure x + x |
| x \* 15 | (x << 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** | **gcc -S -m32 -O1 -fomit-frame-pointer test-sr.c** |
| --- | --- |
| int f(int x) {  return 17\*x;  } | 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 |

Il calcolo di 17\*x viene ridotto a 16\*x+x, equivalente a (x<<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-srl.c** | **Strength reduction (manualmente)** |
| --- | --- |
| void f(int\* v, int n, int k) {  int i;  for (i=0; i<n; i++)  v[i] = **i\*k**;  } | void f(int\* v, int n, int k) {  int i, c = 0;  for (i=0; i<n; i++) {  v[i] = c;  **c += k**;  }  } |

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

#### 5.3.3.3 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 mod s = 0*. Ad esempio, ogni short inizierà ad un indirizzo pari, un int a un indirizzo multipli 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:

| struct S {  char a;  int b;  short c;  char d;  int e;  char f;  }; |  |
| --- | --- |

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.

#### 5.3.3.4 Altre ottimizzazioni

Nel resto di questa dispensa discuteremo altre ottimizzazioni che il programmatore o il compilatore possono sfruttare basate sul particolare design hardware soggiacente. Come casi rilevanti, nel [Paragrafo 6.2.2](#_nlrvnh7cp79l) vedremo inoltre come cambiando l'ordine delle istruzioni macchina di un programma è possibile sfruttare l'architettura interna delle CPU (**pipelining**) per consentire l'esecuzione simultanea di istruzioni diverse. Nel [Paragrafo 7.2.4.1](#_xkweu27zi6lk) vedremo inoltre come cambiando l'ordine con cui si accedono agli indirizzi di memoria si potrà sfruttare la presenza di memorie molto veloci in un sistema di calcolo (**cache**).

### 5.3.4 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 di un programma. Come abbiamo visto nel [Paragrafo 5.2.1](#_hbpg2phe9iqh), 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.

#### 5.3.4.1 Ottimizzare lo spazio richiesto dalle strutture C

Come abbiamo visto nel [Paragrafo 5.2.3.3](#_1fvnl910x3g7), 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 5.2.3.3](#_1fvnl910x3g7) e mostriamo cosa si ottiene riordinano le dichiarazioni dei campi:

| struct S {  int b;  int e;  short c;  char a;  char d;  char f;  }; |  |
| --- | --- |

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

#### 5.3.4.2 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 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)** |
| --- | --- |
| void f() {  if (0) return 10;  return 20;  } | void f() {  return 20;  } |

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:

| **gcc -S -fomit-frame-pointer -m32 test-dce.c** |
| --- |
| f: movl $20, %eax  ret |

### 

### 5.3.5 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.

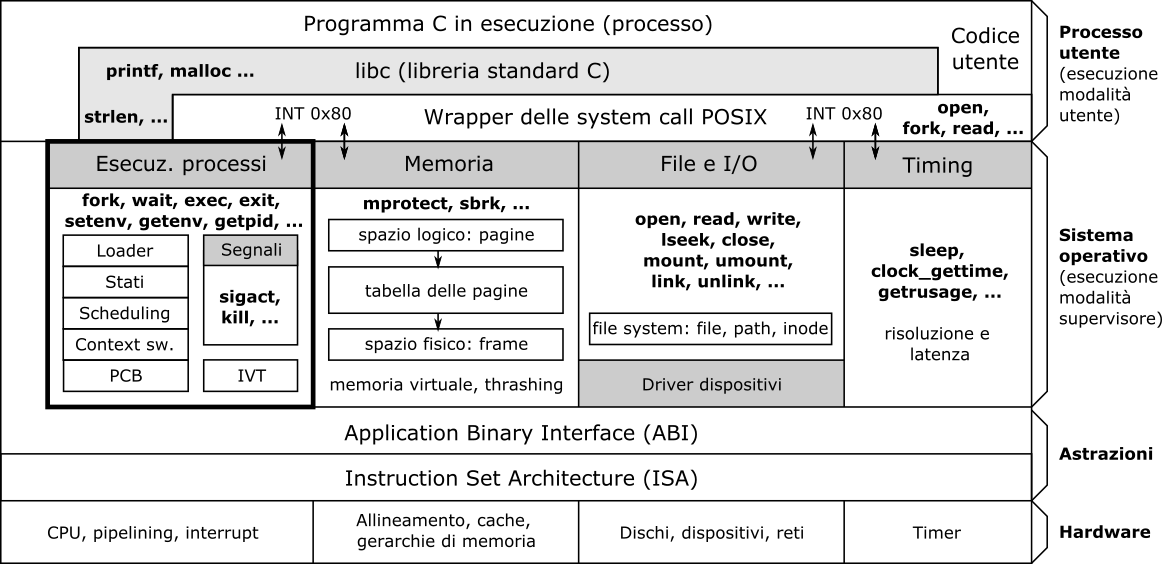
# 

# 6 Come vengono eseguiti i programmi?

## 6.1 Processi

Quando un programma viene eseguito, esso dà luogo a un processo. I **processi** sono semplicemente **programmi in esecuzione**. La loro gestione è uno dei compiti principali di un sistema operativo.

Uno **stesso programma** può essere istanziato in **più processi** che possono coesistere nel sistema. Ogni processo è identificato univocamente da 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.



**Figura 5.1**: panoramica di un sistema di calcolo Linux/x86 come trattato nella dispensa.

Un processo è caratterizzato principalmente da:

* Un’**immagine di memoria** che contiene il codice del programma e i dati da esso manipolati (variabili, blocchi allocati dinamicamente, ecc.). Si veda il [Paragrafo 6.1.1](#_nfatk2cdcocv).
* Lo **stato della CPU** (registri interni, ecc.)
* Un insieme di **risorse in uso** (file aperti, ecc.)
* Un insieme di **metadati** che tengono traccia di 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.)

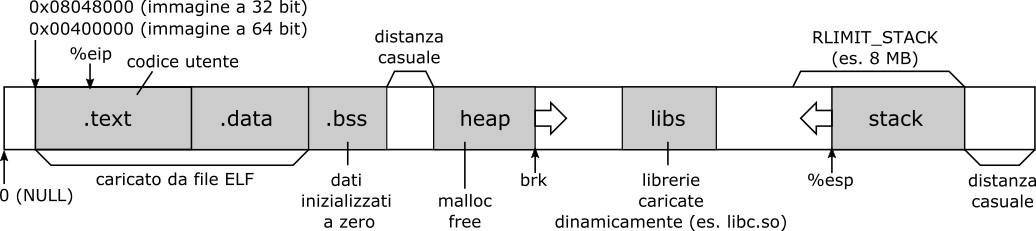
Un processo può essere caricato in memoria e attivato in vari modi:

* 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](#_w6lesegis88)), 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 | | --- | |

### 6.1.1 Immagine di memoria di un processo

La Figura 4.1.1 mostra la tipica organizzazione dell'**immagine di memoria di un processo** ottenuta dopo il caricamento di un file eseguibile (es. in formato ELF). Come vedremo nel [Paragrafo 7.1.3](#_fo77pab1svi2), ogni processo vede questo spazio di memoria come un array di byte con indirizzi che partono da 0 e arrivano a 2b-1, dove b è il numero di bit di un puntatore (es. 32 o 64). I sistemi operativi **isolano** gli spazi di memoria dei processi mediante un meccanismo chiamato **memoria virtuale** affinché un processo non possa accedere allo spazio di memoria di un altro processo in modo malevolo o per via di errori nel codice (bug).



**Figura 4.1.1**: immagine di memoria di un processo tipica di un sistema Linux

Si noti che:

1. la regione **.text** contiene il codice macchina del programma caricato dal file eseguibile (es. in formato ELF) dal modulo di caricamento del kernel del sistema operativo (loader)
2. la regione **.data** contiene dati inizializzati come ad esempio il contenuto dei letterali stringa che appaiono nel programma (es. "hello world") e il contenuto delle variabili globali (esterne alle funzioni oppure locali alle funzioni e dichiarate static). Come la sezione .text, anche la sezione .data viene caricata dal file eseguibile. La porzione che contiene dati in sola lettura come i letterali stringa è denotata con **.rodata** (non mostrata in figura)
3. la regione **.bss** contiene spazio per variabili globali inizializzate a zero
4. a seguire, a una distanza che può variare da un'esecuzione all'altra, vi è la regione **heap** che contiene i blocchi allocati con malloc e calloc. La ragione per randomizzare l'indirizzo di partenza dell'heap è per aumentare la **sicurezza** evitando che un hacker malevolo possa contare su una posizione fissa dei blocchi in memoria. Si noti che l'heap cresce verso indirizzi più alti mano mano che vengono allocati blocchi.
5. la **stack** del programma è nella zona più alta della memoria e cresce verso il basso. Anche l'indirizzo della stack è randomizzato per le stesse ragioni dell'heap. Si noti che la stack ha tipicamente una dimensione massima predefinita (RLIMIT\_STACK).
6. in una posizione intermedia tra heap e stack vi è una regione che contiene le **librerie caricate dinamicamente** dal loader quando servono. Ad esempio, la libc che fornisce le funzioni della libreria standard del linguaggio C.

**Esempio**. Illustriamo con un esempio concreto come le singole parti di un programma C si mappino sull'immagine di memoria del programma in esecuzione. Consideriamo il seguente programma mem.c:

| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  int i[10000000], j = 10;  int main(int argc, char\* argv[]) {  static char \*k = "hello";  int \*p = malloc(sizeof(int));  free(p);  printf("&i=%p|&j=%p|&main=%p|&argc=%p|",&i,&j,&main,&argc);  printf("&k=%p|k=%p|&p=%p|p=%p\n",&k,k,&p,p);  pause();  return 0;  } |
| --- |

Nell'esempio visto, le singole parti vengono collocate come segue nell'immagine di memoria:

1. i → .bss (variabile array globale inizializzato automaticamente a zero)
2. j → .data (variabile globale inizializzata a un valore definito)
3. main → .text (indirizzo iniziale del codice macchina della funzione main)
4. argc, argv → stack (parametri passati secondo l'ABI i386 a 32 bit)
5. k → .data (variabile locale statica inizializzata con l'indirizzo di una stringa)
6. \*k → .data (caratteri del letterale stringa "hello")
7. p → stack (variabile locale allocata in stack)
8. \*p → heap (blocco in heap puntato dall'indirizzo contenuto in p)

In Linux, è possibile esplorare il layout di memoria del processo derivante dall'esempio sopra come segue.

| **Linux:** /proc/[pid]/maps **(immagine del processo)** |
| --- |
| In ambiente Linux è possibile esplorare le varie regioni nell'immagine di memoria di un processo accedendo al file virtuale /proc/[pid]/maps, dove [pid] è il pid del processo da analizzare.  Compiliamo (a 32 bit per mantenere indirizzi piccoli) ed eseguiamo il programma mem.c:   | $ gcc mem.c -o mem -m32  $ ./mem  &i=0x804a060|&j=0x804a02c|&main=0x80484ad|&argc=0xffe56c80|&k=0x804a030|k=0x8048606|&p=0xffe56c6c|p=0xb3c7008 | | --- |   Si noti che il processo stampa gli indirizzi delle variabili e il contenuto delle variabili puntatore. In un altro terminale, otteniamo ora il pid del processo, che è 17339:   | $ ps -e | grep mem[[26]](#footnote-25)  17339 pts/0 00:00:00 mem | | --- |   e visualizziamo il contenuto dello pseudo-file[[27]](#footnote-26) /proc/17339/maps:   | $ cat /proc/17339/maps  08048000-08049000 r-xp 00000000 00:19 55 /home/biar/Documents/temp/mem  08049000-0804a000 r--p 00000000 00:19 55 /home/biar/Documents/temp/mem  0804a000-0804b000 rw-p 00001000 00:19 55 /home/biar/Documents/temp/mem  0804b000-0a670000 rw-p 00000000 00:00 0  0b3c7000-0b3e8000 rw-p 00000000 00:00 0 [heap]  f754d000-f754e000 rw-p 00000000 00:00 0  f754e000-f76f6000 r-xp 00000000 08:01 415168 /lib/i386-linux-gnu/libc-2.19.so  f76f6000-f76f7000 ---p 001a8000 08:01 415168 /lib/i386-linux-gnu/libc-2.19.so  f76f7000-f76f9000 r--p 001a8000 08:01 415168 /lib/i386-linux-gnu/libc-2.19.so  f76f9000-f76fa000 rw-p 001aa000 08:01 415168 /lib/i386-linux-gnu/libc-2.19.so  f76fa000-f76fd000 rw-p 00000000 00:00 0  f771c000-f771f000 rw-p 00000000 00:00 0  f771f000-f7720000 r-xp 00000000 00:00 0 [vdso]  f7720000-f7740000 r-xp 00000000 08:01 415161 /lib/i386-linux-gnu/ld-2.19.so  f7740000-f7741000 r--p 0001f000 08:01 415161 /lib/i386-linux-gnu/ld-2.19.so  f7741000-f7742000 rw-p 00020000 08:01 415161 /lib/i386-linux-gnu/ld-2.19.so  ffe38000-ffe59000 rw-p 00000000 00:00 0 [stack] | | --- |   Osserviamo che alcune regioni sono associate a file (es. /lib/...), ovvero il loro contenuto è derivato da un file, e altre no. Ogni riga contiene, nell'ordine:   1. intervallo di indirizzi della regione di memoria [inizio incluso - fine esclusa] 2. permessi di accesso alla regione[[28]](#footnote-27) 3. offset in byte della regione all'interno del file su cui la regione è mappata 4. codice del dispositivo che contiene il file, o 0 se la regione non è mappata su file 5. codice del file che contiene la regione, o 0 se la regione non è mappata su file 6. nome del file su cui la regione è mappata, [heap] se è l'heap, [stack] se è stack, vuota se la regione è anonima, ecc.   Si noti che tutti gli estremi delle regioni sono multipli di 4096 (4 KB), che come vedremo nel capitolo 7 è la dimensione di una pagina, che rappresenta l'unità elementare da cui tutte le regioni di memoria sono composte.  Vediamo ora di localizzare gli elementi del programma nelle rispettive regioni:   | &i = 0804a060 in 0804a000-0a670000 (.data e .bss)  &j = 0804a02c in 0804a000-0804b000 (.data)  &main = 080484ad in 08048000-08049000 (.text)  &argc = ffe56c80 in ffe38000-ffe59000 (stack)  &k = 0804a030 in 0804a000-0804b000 (.data)  k = 08048606 in 08049000-0804a000 (.rodata, stringa read-only)  &p = ffe56c6c in ffe38000-ffe59000 (stack)  p = 0927c008 in 0b3c7000-0b3e8000 (heap) | | --- |   Si noti che la variabile i parte nella regione .data. Essendo di dimensione 400 MB, la prima parte risiede in .data fino a completare la dimensione di una pagina e il resto (il grosso) risiede nella regione .bss.  Purtroppo nel file maps non vengono nominate le regioni .text, .data, .bss, .rodata. Tuttavia, non è difficile identificarli fra quelle anonime:   * .text ha il permesso di esecuzione e risiede nel file eseguibile (ELF) * .data ha permessi di lettura e scrittura e risiede nel file eseguibile (ELF) * .bss viene prima dell'heap, ha permessi di lettura e scrittura, e non risiede su disco * .rodata ha permesso di sola lettura |

La spiegazione di altre regioni come [vdso] e le librerie dinamiche .so è al di là degli scopi del corso.

## 6**.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.

### 6.2.1 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 sottoparti di un'istruzione. Riportiamo a titolo di esempio una sequenza di stadi, presa in prestito dai classici processori RISC[[29]](#footnote-28), 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 viene dopo l'esecuzione della somma. In architetture reali come i moderni processori Intel si può arrivare a 20 stadi e oltre.

Il calcolo di 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 combinatori 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, assumendo che la frequenza sia costante nel tempo.

### 6.2.2 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 (**hazard**) 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](#_nhh5lys285s3)) 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:



**Hazard sul controllo.** Vi sono varie tecniche hardware per mitigare gli hazard sul controllo, fra cui gli stalli e la predizione del ramo seguito da un salto condizionato. Dal punto di vista del compilatore/programmatore, una tecnica molto efficace è cercare di ridurre i salti condizionati rimpiazzandoli con istruzioni che non alterano il flusso del controllo (program counter): queste istruzioni, come la CMOV, vengono dette **branchless**. Consideriamo il seguente esempio:

testl %eax, %eax

je L

movl %ecx, %eax

L: ...

L'intera sequenza può essere rimpiazzata con una versione che non usa salti condizionati:

testl %eax, %eax

cmovnel %ecx, %eax ← istruzione branchless

...

### 6.2.3 Esecuzione simultanea di più processi

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.

#### 6.2.3.1 Stati di un processo, schedulazione, preemption, time-sharing

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

1. **ready**: il processo è pronto per l'esecuzione, ma è tenuto fermo poiché tutti i core della CPU disponibili sono impegnati nell'esecuzione di altri processi;
2. **running**: il processo è attualmente in esecuzione su un core della CPU;
3. **waiting** (o **sleeping**): il processo è in attesa di un evento, come il completamento di un'operazione di I/O. Ad esempio, un processo entra nello stato waiting quando esegue una chiamata a sistema che richiede la lettura di un blocco di dati da disco. Quando un processo entra in stato waiting, libera il core su cui era in esecuzione in modo che diventi disponibile per un altro processo ready.

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 sè 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.

#### 6.2.3.2 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:

| **#include <unistd.h>**  **pid\_t fork();** |
| --- |
| La chiamata genera un **nuovo processo** figlio ottenuto come **clone** di quello genitore che esegue la fork. In uscita dalla chiamata a fork, sia il processo figlio che quello genitore proseguono l'esecuzione in modo indipendente dall'istruzione che segue la fork. E' possibile distinguere il genitore dal figlio tramite il valore restituito dalla fork. |
| **Parametri**:   * nessuno |
| **Risultato**:   * pid del processo figlio, per il processo genitore * 0 per il processo figlio * -1 in caso di errore, descritto dalla variabile errno |

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:

| #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");  }  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.

#### 6.2.3.3 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.

#### 6.2.3.4 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:

| **#include <sys/types.h>**  **#include <sys/wait.h>**  **pid\_t wait(int\* status\_ptr);** |
| --- |
| Attende la terminazione di un processo figlio |
| **Parametri**:   * puntatore a un buffer in cui viene scritto lo stato di terminazione del processo figlio, oppure NULL se questa informazione non è di interesse. |
| **Risultato**:   * pid del processo figlio terminato * -1 in caso di errore, descritto dalla variabile errno |

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:

| #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) {  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;  } |
| --- |

#### 6.2.3.5 Passaggio dei parametri ai programmi: argc e argv

Come vedremo nel paragrafo 6.2.3.6, è possibile passare un array di stringhe come parametri a un programma. Questo array può essere acceduto dal programma usando gli argomenti del main argc e argv come nel seguente esempio:

| #include <stdio.h>  int main(int argc, char\* argv[]) {  int i;  for (i=0; i<argc; ++i) printf("%s\n", argv[i]);  return 0;  } |
| --- |

Si noti che argv[0] contiene il nome del programma eseguibile stesso, mentre il primo parametro è in argv[1]. Ad esempio, compilando il programma precedente con il nome param, l'invocazione ./param uno due tre genera il seguente output:

| ./param  uno  due  tre |
| --- |

#### 6.2.3.6 Caricamento di programmi (loader): exec

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 importante 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 in generale 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;  } |
| --- |

## 6.3 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.

### 6.3.1 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[[30]](#footnote-29)**) 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 sottocategorie[[31]](#footnote-30):
  1. **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.
  2. **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 tentativo di lettura o scrittura a una zona di memoria a cui il programma non è consentito accedere genera un fault (page fault).
  3. **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 malfunzionamento può generare un abort.

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

#### 6.3.1.1 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 simultaneamente.

#### 6.3.1.2 System call e loro invocazione con int $0x80

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 IA32[[32]](#footnote-31), una chiamata a sistema si realizza mediante l'istruzione INT $128 (o INT $0x80), dopo aver predisposto un certo numero di registri:

| eax | numero identificativo della system call[[33]](#footnote-32) |
| --- | --- |
| ebx | 1° argomento |
| ecx | 2° argomento |
| edx | 3° argomento |
| esi | 4° argomento |
| edi | 5° argomento |
| ebp | 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  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 |
| --- |

#### 6.3.1.3 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 segnalerà 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.

### 6.3.2 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.

#### 6.3.2.1 Definizione segnali

Un segnale è definito:

1. da un **numero** che lo identifica univocamente nel sistema usato
2. dal **pid** del processo a cui deve essere notificato[[34]](#footnote-33).

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 le numerazioni degli interrupt e dei segnali sono del tutto diverse.

Salvo intervento del programmatore per ottenere un risultato diverso, ogni segnalcc e, una volta notificato al processo a cui è rivolto, risulta in uno dei seguenti **comportamenti[[35]](#footnote-34)** predefiniti:

| **Term** | Il processo ricevente termina |
| --- | --- |
| **Core** | Il processo ricevente termina e genera un file chiamato core dump che consiste in un'istantanea dello stato della memoria dei processi 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:

| **Categoria** | **Segnale** | **N[[36]](#footnote-35)** | **Descrizione** | **Ignorabile** | **Gestibile** | **Default** |
| --- | --- | --- | --- | --- | --- | --- |
| Terminazione | SIGTERM | 15 | Terminazione "morbida" in cui il processo ha la facoltà di liberare le proprie risorse. Inviato dal comando kill 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 ricevente | Sì | Sì | **Ign** |
| 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.

#### 6.3.1.2 Invio segnali

Fra le varie **system call** progettate per inviare programmaticamente un segnale a un processo [riportiamo](http://pubs.opengroup.org/onlinepubs/9699919799/):

| **#include <signal.h>**  **int kill(pid\_t pid, int sig);** |
| --- |
| **Parametri**:   * pit\_t pid[[37]](#footnote-36): pid del processo a cui il segnale deve essere inviato. Il mittente è il processo che esegue la kill. * int sig: numero del segnale da inviare, fra quelli listati in <[signal.h](https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/signal.h.html)>. Per chiarezza, si raccomanda l'utilizzo della costante alfanumerica, ad esempio SIGKILL invece di 9. Nel caso speciale sig=0, kill non invia alcun segnale e verifica semplicemente la validità del pid destinatario. |
| **Risultato**:   * 0 in caso di successo * -1 in caso di errore, descritto dalla variabile errno |

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

| **> kill -segnale pid** | **> kill -s segnale pid** |
| --- | --- |

Il comando invia un segnale al pid specificato. Il comando consente di inviare solo alcuni segnali come definito dal [manuale opengroup](http://pubs.opengroup.org/onlinepubs/9699919799/). 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.

#### 6.3.1.4 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](https://pubs.opengroup.org/onlinepubs/9699919799/functions/signal.html), 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](https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/signal.h.html)> * 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.[[38]](#footnote-37) 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.

| #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;  } |
| --- |

#### 6.3.1.3 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.

#### 6.3.1.4 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);[[39]](#footnote-38)** |
| --- |
| **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);[[40]](#footnote-39)** |
| --- |
| **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 precedente 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/ualarm.html) per la discussione della granularità dei timer.

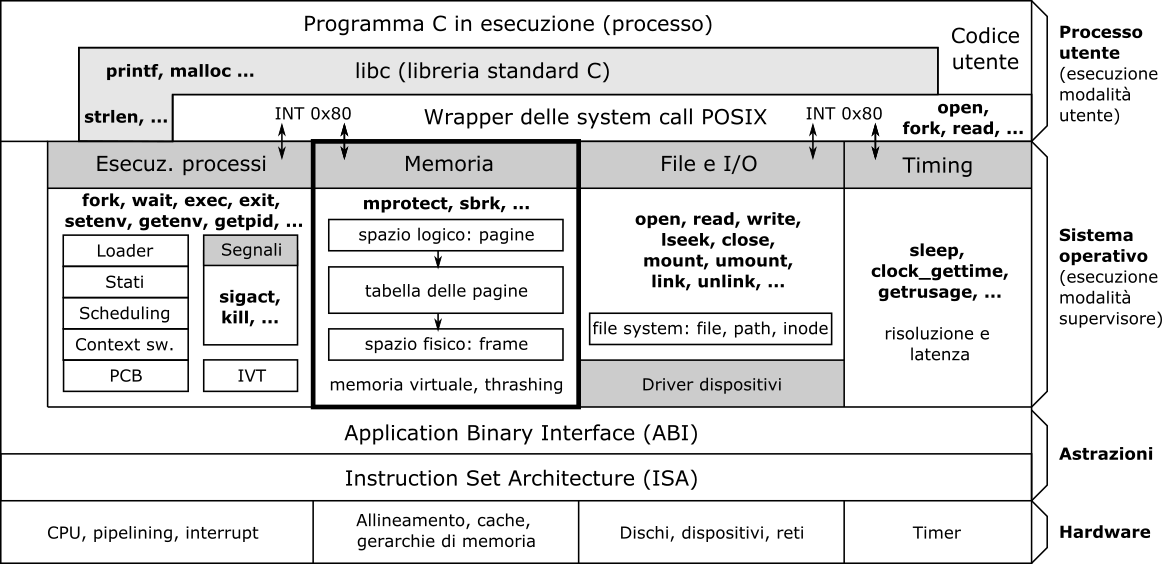
# 

# 

# 7 Come vengono gestiti i dati in memoria centrale?

## 7.1 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 antichi e fondamentali dell'informatica.



**Figura 6.1**: panoramica di un sistema di calcolo Linux/x86 come trattato nella dispensa.

### 7.1.2 Unità di misura della quantità di memoria

A seconda dello scenario, la memoria si misura in bit o byte. Ad esempio, la dimensione di un registro o di un bus si misura in bit, mentre la dimensione della memoria si misura in byte o suoi multipli:

| **Potenze di due (es. RAM, cache) - binario[[41]](#footnote-40)** | | | **Potenze di dieci (es. dischi) - decimale** | | |
| --- | --- | --- | --- | --- | --- |
| KiB | kibibyte | 210 byte | KB | kilobyte | 103 byte |
| MiB | mibibyte | 220 byte | MB | megabyte | 106 byte |
| GiB | gibibyte | 230 byte | GB | gigabyte | 109 byte |
| TiB | tebibyte | 240 byte | TB | terabyte | 1012 byte |
| PiB | pebibyte | 250 byte | PB | petabyte | 1015 byte |
| EiB | exbibyte | 260 byte | EB | exabyte | 1018 byte |
| ZiB | zebibyte | 270 byte | ZB | zettabyte | 1021 byte |
| YiB | yobibyte | 280 byte | YB | yottabyte | 1024 byte |

In genere, i costruttori di dischi usano la notazione KB, MB, ecc. per riferirsi a potenze di 10. Tuttavia, questa stessa notazione viene storicamente usata per riferirsi a potenze di due quando ci si riferisce alle memorie RAM o cache ([Paragrafo 7.2.1](#_nhh5lys285s3)). Per evitare ambiguità, si suggerisce di usare la notazione KiB, MiB, ecc. per riferirsi a potenze di due.

### 7.1.3 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[[42]](#footnote-41) 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 propri 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**.

#### 7**.**1**.**3**.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 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.

#### 7**.**1**.**3**.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 **mancanza di informazioni sulle allocazioni/deallocazioni future**. Se un allocatore potesse attendere di aver accumulato sufficienti 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 (gestione on-line).

#### 7**.**1**.**3**.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.

### 7.1.4 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 maliziosamente[[43]](#footnote-42) 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.

### 7.1.5 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.

#### 7**.**1**.**5**.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.

**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.

#### 7**.**1**.**5**.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.

Nei sistemi POSIX è possibile ottenere la dimensione in byte di una pagina usando la chiamata a sistema sysconf come segue:

| #include <stdio.h>  #include <unistd.h> // sysconf  int main(void) {  printf("Dimensione pagina: %ld byte.\n", sysconf(\_SC\_PAGESIZE));  return 0;  } |
| --- |

**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**.

#### 7**.**1**.**5**.3 Come avviene il mapping degli indirizzi logici su quelli fisici?**

Supponiamo di avere uno spazio virtuale di 2v byte (ad esempio con v=32 o v=64), pagine di 2d byte (ad esempio con d=12, cioè pagine da 4 KB) e spazio fisico di 2r 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:



#### 7**.**1**.**5**.**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 232 byte (4 GB ~ 4 miliardi di byte), mentre su una piattaforma a 64 bit è di 264 (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, una parte delle pagine viene tenuta su disco in un'area chiamata area di **swap** e 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 il bit di validità vale zero, si possono avere due possibilità:

1. la pagina non è allocata al processo, il che porta all'invio di un segnale di **segmentation fault** al processo stesso, che di default ne provoca la terminazione
2. la pagina è allocata al processo, ma risiede su disco nell'area di swap. In questo caso viene generato un **page fault** che innesca una catena di eventi:
   1. il kernel seleziona un **frame vittima**, tipicamente quello che non è stato acceduto da più tempo (politica least-recently used, o LRU), il cui contenuto viene salvato nell'area di swap (**swap-out**)
   2. la pagina richiesta viene caricata dall'area di swap nel frame vittima (**swap-in**)
   3. la tabella delle pagine viene modificata facendo puntare la pagina richiesta al frame vittima e settando il bit di validità a 1
   4. la computazione riprende rieseguendo l'istruzione che ha provocato il page fault

Si noti che un page fault ha un costo notevole, richiedendo due accessi a disco per salvare il frame vittima e caricare la pagina richiesta. Come tale, deve essere un evento raro. Affinché questo avvenga, è necessario che vi siano sufficienti frame liberi allocati al processo quanto è la grandezza del **working set**, ovvero l'insieme delle pagine correntemente in uso attivo al processo. Se questo non avviene, si rischia il fenomeno del **thrashing**, ovvero una situazione in cui il sistema spende più tempo nello swapping su disco che per effettuare progresso nella computazione. Il thrashing si combatte allocando più frame fisici al processo (sottraendoli eventualmente ad altri processi), oppure terminando il processo stesso. In generale si entra in thrashing se tutti i processi correntemente nel sistema hanno una working set maggiore dell'insieme dei frame ad essi allocati.

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.

Sebbene sia la scelta di elezione nella maggior parte dei sistemi, ha anche degli **svantaggi**, fra cui:

1. si presta al fenomeno del thrashing
2. il sistema di gestione di memoria diventa più complesso
3. sistemi embedded con risorse limitate potrebbero non avere un sistema di memoria che consente la paginazione con bit di validità

**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 220-1.



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.

#### 7.1.5.5 Paginazione su richiesta

La paginazione su richiesta (**on-demand paging**) è uno schema pigro (**lazy**) in cui inizialmente tutte le pagine sono su disco e vengono portate nei frame disponibili al processo solo la prima volta che vengono accedute. Questo sistema è ampiamente diffuso e ha il vantaggio che non è necessario fare swap-in all'avvio del processo di pagine che non verranno mai usate dal processo.

### 7.1.6 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[[44]](#footnote-43). Queste primitive partizionano lo spazio heap, controllato principalmente mediante la system call sbrk[[45]](#footnote-44), 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:



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:
  1. *first-fit*: viene scelto il primo blocco libero incontrato;
  2. *best-fit*: viene scelto il più piccolo blocco libero sufficiente a contenere il payload richiesto.

Se non vi sono blocchi liberi sufficientemente grandi, espande l'heap 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.

## 7.2 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 in 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 nanosecondi[[46]](#footnote-45). In altre parole, un'apparentemente innocua istruzione movl $1, (%eax) costerebbe in termini di tempo 100 volte più che movl $1, %eax.

### 7.2.1 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).



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.

### 7.2.2 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.

### 7.2.3 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, questi due princìpi 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.

### 7.2.4 Esempi

Come osservato precedentemente, 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. Vediamo nel [Paragrafo 7.2.4.1](#_xkweu27zi6lk) un esempio in cui l'intervento accorto del programmatore consente il miglioramento delle prestazioni. Vediamo poi come analizzare i cache miss generati da una sequenza di accessi a memoria.

#### 7.2.4.1 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** |
| --- | --- |
| 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;  } | 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;  } |
| 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/sizeof(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.

#### 7.2.4.2 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.



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.

### 7.2.5 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.



### 7.2.6 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.

### 7.2.7 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.

# 

# 8 Come vengono gestiti i file?

Il **file system** è uno dei componenti principali di un sistema operativo. Esso gestisce la memoria esterna, uno dei blocchi più in basso nella [gerarchia di memoria](#_d08tvc72r2wk) caratterizzati da grandi dimensioni e tempi di accesso alti.

# 

**Figura 7.1**: panoramica di un sistema di calcolo Linux/x86 come trattato nella dispensa.

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.

**8.1 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.

Un file system è tipicamente organizzato come un **albero** in cui i nodi interni sono directory e le foglie sono directory o file. I file sono collezioni di dati caratterizzate principalmente da un nome, da una dimensione e da permessi di accesso che stabiliscono chi ha accesso al file e se può leggere, scrivere, o eseguire il file (nel caso sia un file che contiene codice). Una **directory** è un contenitore di altre directory e file. Come i file, hanno un nome e permessi di accesso.

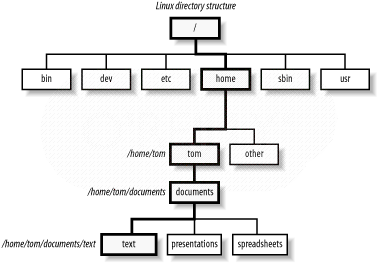
I nomi dei file hanno spesso delle **estensioni** della forma .estensione, che descrivono la natura del loro contenuto. Ad esempio, l'estensione .c denota file di testo scritti in C (es. hello.c), l'estensione .jpg indica file binari che contengono immagini (es. [foto-montagna.jpg](http://1.bp.blogspot.com/_Nx4_AyFzPw0/TMs0JONbGHI/AAAAAAAACoI/QoY1HRCeyqs/s320/linguaccia.jpg)), ecc.

### 8.1.1 L’albero delle directory

Tutti i file accessibili in un sistema UNIX-like (\*nix) come Linux e MacOS X sono organizzati in un **singolo albero complessivo** di file e directory. La **directory radice**, denotata da /, contiene tutti i dati memorizzati in forma permanente nel sistema sotto forma di file e sottodirectory. La struttura ad albero è il risultato di un collage di vari **file system**, che sono strutture dati dove sono memorizzati file, directory, ecc. e che, come vedremo nel [Paragrafo 8.2.3](#_llmtfqa8op1), vengono **montati** sulla struttura ad albero, determinandone la forma complessiva.

In questa dispensa useremo come caso di studio i sistemi [UNIX](http://it.wikipedia.org/wiki/Unix) e [UNIX-like](http://it.wikipedia.org/wiki/Unix-like) (Linux, MacOS X, ecc.), denotati con l'abbreviazione \*nix.

L'organizzazione tipica delle directory di un sistema \*nix, chiamata [Filesystem Hierarchy Standard](http://it.wikipedia.org/wiki/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 \*nix sono tipicamente case-sensitive, pertanto, ad esempio, /home/tom/documents/text è diverso da /home/tom/documents/Text.

### 8.1.2 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. |

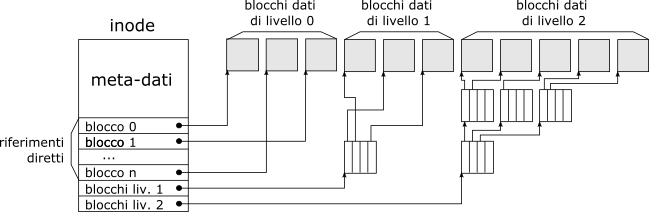
## 8.2 Organizzazione di un file system

### 8.2.1 Inode e blocchi

Nei sistemi Linux e in generale in quelli derivati da UNIX, ogni file, directory e dispositivo collegato al sistema ha associata una struttura chiamata **inode** che viene caricata da disco e tenuta in memoria in un array. Questa gestione elegante consente di trattare in modo uniforme file, directory e dispositivi. L'inode contiene meta-dati e non i dati effettivi, che nel caso dei file risiedono in **blocchi** tutti della stessa dimensione allocati su disco. Un inode contiene tipicamente:

* **Numero dell'inode**: indice nella tabella degli inode
* **Tipo di file**: regolare, directory, ecc.
* **Permessi del file**: lettura, scrittura, esecuzione
* **Numero di hard link al file**: vedi [Paragrafo 8.5.2](#_txda06f2b6i1)
* **User ID**: utente proprietario del file
* **Group ID**: gruppo proprietario del file
* **Dimensione del file**
* **Tempo di ultimo accesso al file**
* **Tempo di ultima modifica al file**
* **Tempo di ultima modifica all'inode**
* **Riferimenti diretti a blocchi**: indici dei primi blocchi del file
* **Riferimento indiretto singolo a blocchi**: indice di blocco che contiene indici a blocchi dati
* **Riferimento indiretto doppio a blocchi**: indice di blocco che contiene indici a blocchi che contengono a loro volta indici a blocchi dati

Si noti che **un inode non contiene il nome del file**, che risiede invece nei blocchi dati delle **directory, viste come file speciali** che contengono coppie (nome, numero inode). In Figura 8.2.1 mostriamo l'organizzazione di un inode che punta ai blocchi dati di un file di grandi dimensioni. Si noti che, poiché i dati sono allocati a multipli di dimensione dei blocchi, il file più piccolo occuperà un blocco. File piccoli usano solo riferimenti diretti. File di dimensioni più grandi usano riferimenti indiretti.



**Figura 8.2.1**: organizzazione inode e collegamenti ai blocchi dati

Assumiamo come caso realistico blocchi di 4096 byte. I blocchi sono di due tipi: **blocchi dati**, che contengono i dati del file, e **blocchi di indici**, che contengono indici di altri blocchi. Ogni indice di blocco richiede tipicamente 32 bit, ovvero 4 byte. Assumiamo come scenario plausibile di avere nell'inode:

* 12 riferimenti diretti a blocchi dati
* un indice di un blocco di indici che puntano a blocchi dati (**indirezione singola**)
* un indice di un blocco di indici che puntano ad altri blocchi di indici che puntano a loro volta a blocchi dati (**indirezione doppia**).

Ogni blocco di indici contiene tipicamente fino a 4096/4=1024 riferimenti a blocchi. In totale, con questo schema si possono avere file grandi fino a (12+1024+1024\*1024)\*4096=4299210752 byte, quindi leggermente oltre i 4 GiB (4294967296 byte). Con uno schema di inode multilivello che prevede **indirezione tripla** si raggiungono dimensioni di file di oltre 16 TiB ciascuno.

**Esempi**. Si ha che:

* un **file di 256 byte** occupa 1 blocco per un totale di 4096 byte
* un **file di 16000 byte** occupa 4 blocchi per un totale di 16384 byte
* un **file di 4 MiB** richiede 4194304 byte per un totale di 1024 blocchi dati suddivisi in 12 blocchi di livello 0, 1012 blocchi di livello 1 e 1 blocco di indici. Il totale occupato su disco è di 1025 blocchi, vale a dire 1025\*4096=4198400 byte
* un **file di 813 MiB** richiede 852492288 byte per un totale di 208128 blocchi dati suddivisi in 12 blocchi di livello 0, 1024 blocchi di livello 1 e 207092 blocchi di livello 2. Il totale occupato dai blocchi di indici è 1+1+203 = 205. Il totale occupato su disco è pertanto di 208228+205=208433 blocchi, vale a dire 208433\*4096 = 853741568 byte

Si noti che la minima allocazione possibile è a livello di blocco ed è sfavorevole in termini di uso ottimale dello spazio su disco nel caso in cui il file system contenga molti file di piccole dimensioni, inferiori alla dimensione di un blocco dati.

E' possibile visualizzare il numero dell'inode di un file con il comando ls -i:

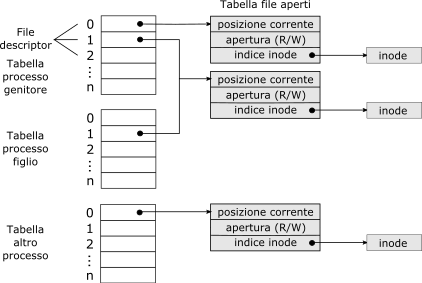
| $ ls /usr -ila  total 324  786433 drwxr-xr-x 11 root root 4096 Dec 20 14:18 .  2 drwxr-xr-x 17 root root 4096 Dec 20 18:26 ..  932223 drwxr-xr-x 4 root root 4096 Dec 19 21:15 avr  787352 drwxr-xr-x 5 root root 98304 Dec 20 14:17 bin  786434 drwxr-xr-x 411 root root 36864 Dec 20 14:17 include  787356 drwxr-xr-x 208 root root 135168 Dec 20 14:17 lib  807469 drwxr-xr-x 14 root root 20480 Dec 19 20:59 lib32  789017 lrwxrwxrwx 1 root root 3 Nov 19 23:24 lib64 -> lib  789028 drwxr-xr-x 11 root root 4096 Sep 30 09:46 local  789039 lrwxrwxrwx 1 root root 3 Nov 19 23:24 sbin -> bin  787358 drwxr-xr-x 230 root root 4096 Dec 19 22:23 share  789067 drwxr-xr-x 2 root root 4096 Aug 28 21:58 src  931438 drwxr-xr-x 3 root root 4096 Dec 19 21:15 x86\_64-pc-linux-gnu  $ |
| --- |

La prima colonna rappresenta il numero di inode dei file, directory, ecc. E' possibile esplorare alcuni dei campi più rilevanti di un inode con il comando stat.

**Esempio**. Mostriamo un esempio di uso del comando stat su piattaforma Linux Arch 64 bit.

| $ stat /bin/ls  File: /bin/ls  Size: 137776 Blocks: 272 IO Block: 4096 regular file  Device: 802h/2050d Inode: 799795 Links: 1  Access: (0755/-rwxr-xr-x) Uid: (0/root) Gid: (0/root)  Access: 2019-12-19 20:57:23.000000000 +0100  Modify: 2019-11-12 12:00:56.000000000 +0100  Change: 2019-12-19 20:57:23.510021617 +0100  Birth: 2019-12-19 20:57:23.510021617 +0100 |
| --- |

L'output di stat ci dice che il file è regolare (non directory, dispositivo, o altro) ha dimensione 137776 byte, occupa 272 blocchi di 4096 ciascuno, ha numero di inode 799795 un solo link. Seguono altre informazioni dell'inode.



**Figura 8.2.2**. Tabella dei descrittori di file e tabella file aperti

### 8.2.2 Tabella dei descrittori di file di un processo

Ogni processo ha una propria tabella, chiamata **tabella dei descrittori di file**, le cui entry sono puntatori a un'altra **tabella di file aperti** che contengono varie informazioni (Figura 8.2.2[[47]](#footnote-46)), fra cui:

* la posizione corrente nel file
* la modalità di apertura (in lettura, scrittura, ecc.)
* l'indice dell'inode del file

Gli indici delle entry della tabella dei descrittori di file sono chiamati **file descriptor** e sono usati dalle system call per riferirsi a un particolare canale di dati collegato con un file, una directory, un dispositivo, ecc. Le prime tre entry sono automaticamente aperte all'avvio del processo e corrispondono ai canali di standard input (0), output (1), ed error (2).

E' possibile esplorare le entry della tabella dei file aperti di un processo con pid x corrispondenti a file aperti esplorando la pseudo-directory /proc/x/fd come illustrato nel seguente esempio.

**Esempio**.

| $ ls -l /proc/21569/fd  lrwx------ 1 studente studente 64 Dec 21 04:50 0 -> /dev/pts/0  lrwx------ 1 studente studente 64 Dec 21 04:50 1 -> /dev/pts/0  lrwx------ 1 studente studente 64 Dec 21 04:50 2 -> /dev/pts/0  l-wx------ 1 studente studente 64 Dec 21 04:50 3 -> /home/studente/Desktop/miofile.txt  $ |
| --- |

Si noti che i primi tre descrittori di file (fd) sono relativi allo standard input (0), standard output (1) e standard error (2) e tutti e tre fanno capo al dispositivo /dev/pts/0 che rappresenta il terminale in cui gira la shell che ha aperto il processo. C'è un ulteriore descrittore di file in uso, il 3, che si riferisce a un file regolare correntemente aperto dal processo.

## 8.3 Mount e unmount di file system sull'albero delle directory

I file system sono visti come dispositivi e sono tipicamente rappresentati come file speciali nella directory /dev, ad esempio /dev/sda1:

| $ file /dev/sda1  /dev/sda1: block special (8/1)  $ |
| --- |

Per poter rendere fruibile un file system a programmi e utenti è necessario che questo sia **montato** sull'albero delle directory. Per vedere l'elenco dei file system montati e su quale directory (**mount point**), si può usare il comando df:

| $ df  Filesystem Size Used Avail Use% Mounted on  dev 2.0G 0 2.0G 0% /dev  run 2.0G 820K 2.0G 1% /run  /dev/sda2 29G 12G 16G 42% /  tmpfs 2.0G 8.0K 2.0G 1% /dev/shm  tmpfs 2.0G 0 2.0G 0% /sys/fs/cgroup  tmpfs 2.0G 46M 1.9G 3% /tmp  /dev/sda1 488M 55M 398M 13% /boot  tmpfs 394M 24K 394M 1% /run/user/1000  $ |
| --- |

Ad esempio, il file system /dev/sda2 è montato sulla radice dell'albero delle directory /.

Per **montare** un file system si può usare il **comando** mount dispositivo mount\_point come nel seguente esempio, il cui mount point è /home/studente/Desktop/temp:

| $ mkdir /home/studente/Desktop/temp  $ touch /home/studente/Desktop/temp/prova  $ sudo mount /dev/sda2 /home/studente/Desktop/temp  $ ls /home/studente/Desktop/temp  bin dev home lib64 mnt proc run srv tmp var  boot etc lib lost+found opt root sbin sys usr  $ |
| --- |

Si noti che per eseguire mount servono privilegi da amministratore (sudo). Quando un file system è montato su una directory, il **mount point si riferisce alla radice del file system**, anch'esso organizzato ad albero. Si noti che **solo un file system alla volta** può essere montato su una directory. Il montaggio non è permanente e viene annullato nel momento del riavvio o spegnimento della macchina. Eventuali file/directory preesistenti nel mount point non appariranno più fino al momento in cui il file system del dispositivo non verrà smontato. Ad esempio, il file prova creato non è più visibile.

Per **smontare** un file system si può usare il **comando** umount mount\_point, che ristabilisce la situazione precedente al montaggio del file system sul mount\_point. Con riferimento all'esempio precedente:

| $ sudo umount /home/studente/Desktop/temp  $ ls /home/studente/Desktop/temp  prova  $ |
| --- |

Si noti che per eseguire umount servono, come per mount, privilegi da amministratore (sudo)

## 8.4 Operazioni su file e directory

Nel [Paragrafo 4.2.3](#_er65d0utpnmh) abbiamo trattato funzioni per la manipolazione di file contenute nella libreria standard C (libc). Nel resto di questo capitolo ci occuperemo delle **chiamate a sistema** su cui si basano le funzioni di accesso al file system della libc. Queste chiamate a sistema offrono funzionalità più di base, ad esempio permettono solo la scrittura/lettura byte a byte, ma anche maggiore ricchezza nell'accesso al file system. Ad esempio, permettono di esplorare gli attributi di un file, iterare su una directory, impostare i privilegi di accesso ecc., operazioni non incluse nella libc.

### 8.4.1 Apertura e chiusura: open e close

La prima operazione per poter accedere a un file è la sua apertura con open:

| **#include <fcntl.h>**  **int open(const char \*pathname, int flags);**  **int open(const char \*pathname, int flags, mode\_t mode);** |
| --- |
| **Parametri**:   * pathname: percorso e nome del file da aprire * flags: modalità di apertura del file (si veda man 2 open) * mode (opzionale): privilegi di accesso file (si veda man 2 open) |
| **Risultato**:   * in caso di successo, descrittore intero che rappresenta il canale aperto verso il file * in caso di errore, -1 |
| **Esempio:** creazione di un file nella directory corrente  #include <fcntl.h> // per open  #include <unistd.h> // per write e close  #include <assert.h> // per assert  #include <string.h> // per strlen  int main() {  // apre file in sola scrittura dopo averlo svuotato;  // crea file se non esiste  int fd = open("miofile.txt", O\_CREAT | O\_WRONLY | O\_TRUNC, 0640);  assert(fd != -1);  char \*str = "hello world";  write(fd, str, strlen(str));  close(fd);  return 0;  } |

Il descrittore di file restituito da open è **sempre quello più piccolo** corrispondente alla prima entry non in uso della tabella dei file del processo.

In modo analogo a fclose, la chiusura di un file si effettua con close:

| **#include <unistd.h>**  **int close(int fd);** |
| --- |
| **Parametri**:   * fd: descrittore del canale da chiudere |
| **Risultato**:   * 0 in caso di successo * -1 in caso di errore; viene inoltre settato errno al codice di errore generato |
| **Esempio:** si veda quello mostrato per open |

### 8.4.2 Scrittura e lettura: write e read

La forma più semplice di accesso file è quella sequenziale in cui i byte vengono acceduti nell'ordine in cui appaiono nel file. In modo analogo a fwrite, la scrittura di file si effettua con write:

| **#include <unistd.h>**  **ssize\_t write(int fd, const void \*buf, size\_t count);** |
| --- |
| **Descrizione**: la chiamata a sistema scrive il contenuto di un buffer su un canale aperto e sposta in avanti la posizione corrente nel file del numero di byte scritti |
| **Parametri**:   * fd: descrittore del canale aperto su cui scrivere * buf: puntatore al buffer da scrivere sul canale * count: numero di byte da scrivere |
| **Risultato**:   * numero di byte scritti, in caso di successo * -1 in caso di errore; viene inoltre settato errno al codice di errore generato |
| **Esempio:** si veda quello mostrato per open |

In modo analogo a fread, la lettura di file si effettua con read:

| **#include <unistd.h>**  **ssize\_t read(int fd, void \*buf, size\_t count);** |
| --- |
| **Descrizione**: la chiamata a sistema legge da un canale aperto dei byte e li scrive in un buffer, spostando in avanti la posizione corrente nel file del numero di byte letti |
| **Parametri**:   * fd: descrittore del canale aperto da cui leggere * buf: puntatore al buffer in cui scrivere i byte letti dal canale * count: numero di byte da scrivere |
| **Risultato**:   * numero di byte letti, in caso di successo * -1 in caso di errore; viene inoltre settato errno al codice di errore generato |
| **Esempio:** lettura di un file dalla directory corrente (usare prima l'esempio di open)  #include <fcntl.h> // per open  #include <unistd.h> // per read e close  #include <assert.h> // per assert  #include <stdio.h> // per printf e fwrite  int main() {  // apre file in sola lettura  int fd = open("miofile.txt", O\_RDONLY);  assert(fd != -1);  char str[512];  ssize\_t r = read(fd, str, sizeof(str));  close(fd);  printf("letti %lu byte: ", r);  fwrite(str, sizeof(char), r, stdout);  printf("\n");  return 0;  } |

### 8.4.3 Accesso casuale: lseek

Per accedere in modo non sequenziale a un canale (cosa non possibile ad esempio per i canali di I/O collegati con il terminale), è possibile utilizzare la system call lseek, che sposta la posizione corrente nel punto dove si intende scrivere o leggere. La corrispondente funzione libc è fseek (non trattata in questa dispensa: si veda man fseek).

| **#include <unistd.h>**  **off\_t lseek(int fd, off\_t offset, int whence);** |
| --- |
| **Parametri**:   * fd: descrittore del canale aperto su cui lavorare * offset: spiazzamento rispetto al punto di riferimento specificato da whence * whence: punto di riferimento, che può essere inizio file, posizione corrente, o fine del file (si veda man 2 lseek) |
| **Risultato**:   * nuova posizione misurata come numero di byte rispetto all'inizio del file in caso di successo * -1 in caso di errore; viene inoltre settato errno al codice di errore generato |
| **Esempio:** lettura degli ultimi 5 byte da un file dalla directory corrente (usare prima l'esempio di open)  #include <fcntl.h> // per open  #include <unistd.h> // per read e close  #include <assert.h> // per assert  #include <stdio.h> // per printf e fwrite  int main() {  // apre file in sola lettura  int fd = open("miofile.txt", O\_RDONLY);  assert(fd != -1);  char str[512];  off\_t pos = lseek(fd, -5, SEEK\_END);  ssize\_t r = read(fd, str, sizeof(str));  close(fd);  printf("letti %lu byte dalla posizione %lu: ", r, pos);  fwrite(str, sizeof(char), r, stdout);  printf("\n");  return 0;  } |

E' anche possibile leggere la posizione corrente in un file con lseek usando come offset 0 e come whence SEEK\_CURR. La corrispondente funzione libc per questa funzionalità è ftell (non trattata in questa dispensa: si veda man ftell).

### 8.4.4 Lavorare con le directory

Per aprire una directory è possibile usare la funzione opendir:

| **#include <dirent.h>**  **DIR \*opendir(const char \*dirname);** |
| --- |
| **Parametri**:   * dirname descrittore del canale aperto su cui lavorare |
| **Risultato**:   * puntatore a struttura che rappresenta la directory aperta * NULL in caso di errore; viene inoltre settato errno al codice di errore generato |
| **Esempio:** mostriamo come iterare su una directory per cercare un nome di file  #include <dirent.h> // DIR, struct dirent, opendir, readdir  #include <stdio.h> // printf  #include <string.h> // strcmp  int main(int argc, char \*argv[]) {  int found = 0;  DIR \*dirp;  struct dirent\* dp;  if (argc < 3) return (printf("usage: findname dirname name\n"),-1);  dirp = opendir(argv[1]);  if (dirp == NULL) return -1;  while ((dp = readdir(dirp)) != NULL) {  if (strcmp(dp->d\_name, argv[2]) == 0) {  found = 1;  break;  }  }  printf("%s\n", found ? "trovato" : "non trovato");  closedir(dirp);  return 0;  } |

Per chiudere una directory si usa la chiamata closedir:

| **#include <dirent.h>**  **int closedir(DIR \*dirp);** |
| --- |
| **Parametri**:   * dirp descrittore della directory da chiudere |
| **Risultato**:   * 0 in caso di successo * -1 in caso di errore; viene inoltre settato errno al codice di errore generato |
| **Esempio:** si veda quello di opendir |

Per leggere il contenuto di una directory, si può usare la chiamata readdir:

| **#include <dirent.h>**  **struct dirent \*readdir(DIR \*dirp);** |
| --- |
| **Parametri**:   * dirp descrittore della directory da leggere |
| **Risultato**:   * puntatore alla successiva struttura che contiene informazioni sulla entry letta in caso di successo.  | **struct dirent** | | --- | | **Attributi:**   * ino\_t d\_ino: numero dell'inode * char d\_name[]: nome del file o directory contenuto in dirp |  * NULL in caso di errore; viene inoltre settato errno al codice di errore generato |
| **Esempio:** si veda quello di opendir |

## 8.5 Link

E' possibile creare degli **alias** per un file chiamati link. Un link è utile per far apparire un file in directory diverse senza dovere duplicare i dati. Le modifiche al file fatte usando il link sono visibili sia accedendo al file originale che tramite il link.

Vi sono due tipi di link: simbolici (o soft) e hard.

### 8.5.1 Symbolic link

E' un file che contiene il percorso del file originale. Esso può risiedere anche su un file system diverso. Se il file originale linkato viene spostato in un'altra directory, il symbolic link diviene indefinito. Cancellare un symbolic link non cancella il file originale. Un symbolic link ha un inode diverso da quello del file linkato.

**Esempio**. Il comando ln -s target link\_name crea un symbolic link con pathname link\_name al file originale già esistente con pathname target.

| $ touch /home/studente/miofile.txt  $ ls -i /home/studente/miofile.txt  **1193992** /home/studente/miofile.txt  $ **ln -s /home/studente/miofile.txt /home/studente/Desktop/miofile.txt**  $ ls -li /home/studente/Desktop/miofile.txt  **1328271** lrwxrwxrwx 1 studente studente 26 Dec 21 03:40 Desktop/miofile.txt -> /home/studente/miofile.txt  $ |
| --- |

Si osservi che i symbolic link vengono visualizzati dal comando ls -l mediante la dicitura link\_name -> target come appare nell'esempio precedente.

### 8.5.2 Hard link

E' a tutti gli effetti un mirror del file originale, indistinguibile da esso. Diversamente da un symbolic link, deve risiedere nello stesso file system dell'originale. Il file viene cancellato quando tutti i hard link a quel file sono stati eliminati. Tutti i hard link, compreso l'originale, hanno lo **stesso inode**, pertanto rappresentano a tutti gli effetti nomi diversi del medesimo file.

#### 8.5.2.1 Creazione di link: link

Il comando ln target link\_name crea un hard link con pathname link\_name del file esistente con pathname target.

| $ touch /home/studente/miofile.txt  $ ls -i /home/studente/miofile.txt  **1193992** /home/studente/miofile.txt  $ **ln /home/studente/miofile.txt /home/studente/Desktop/miofile.txt**  $ ls -i /home/studente/Desktop/miofile.txt  **1193992** /home/studente/Desktop/miofile.txt  $ |
| --- |

dove ln è un comando per creare link che prende come primo argomento il file da linkare e come secondo argomento il link da creare. Si noti che i pathname /home/studente/miofile.txt e /home/studente/Desktop/miofile.txt hanno il medesimo inode number, quindi sono alias indistinguibili del file miofile.txt. Nell**'inode c'è un contatore del numero di hard link al file**: quando questo contatore raggiunge 0 il file viene eliminato insieme al suo inode.

| $ rm /home/studente/miofile.txt  $ ls -i /home/studente/Desktop/miofile.txt  **1193992** /home/studente/Desktop/miofile.txt  $ |
| --- |

Si noti che il link creato precedentemente rimane come link al file. Eliminato anche questo, il file stesso viene eliminato.

La system call link serve per creare nuovi nomi (hard link) per file, directory, dispositivi, ecc., esistenti:

| **#include <unistd.h>**  **int link(const char \*oldpath, const char \*newpath);** |
| --- |
| **Parametri**:   * oldpath: percorso e nome del file da linkare * newpath: percorso e nome del link da creare |
| **Risultato**:   * 0, in caso di successo * -1 in caso di errore; viene inoltre settato errno al codice di errore generato |
| **Esempio:** implementazione di un comando ln semplificato.  #include <unistd.h> // per link  #include <stdio.h> // per printf e perror  int main(int argc, char\* argv[]) {  if (argc < 3) return printf("missing arguments\n"), -1;  int res = link(argv[1], argv[2]);  if (res == -1) perror("error in creating link: ");  return res;  } |

#### 8.5.2.2 Eliminazione di link: unlink

Per eliminare un hard link, si può usare il comando unlink, che rappresenta una versione semplificata del comando rm ([Appendice B.2.8](#_nbbbywdqvk4e)). La sintassi è unlink pathname che elimina l'hard link pathname.

Per eliminare programmativamente un nome (hard link) da un file system, si usa la system call unlink.

**Esempio**. mostriamo come implementare il comando unlink:

| **#include <unistd.h>**  **int unlink(const char \*pathname);** |
| --- |
| **Parametri**:   * pathname: percorso e nome del file da eliminare |
| **Risultato**:   * 0, in caso di successo * -1 in caso di errore; viene inoltre settato errno al codice di errore generato |
| **Esempio:** implementazione di un comando unlink semplificato.  #include <unistd.h>  #include <stdio.h>  int main(int argc, char\* argv[]) {  if (argc < 2) return printf("pathname mancante\n"), -1;  int res = unlink(argv[1]);  if (res == -1) perror("error in deleting name: ");  return res;  } |

# 

# 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.

[Balch 2003] Mark Balch. Complete Digital Design: A Comprehensive Guide To Digital Electronics And Computer System Architecture. McGraw-Hill Professional. p. 430. ISBN 978-0-07-140927-8, 2003.

[Dijkstra 1968] Edsger W. Dijkstra. Go To Statement Considered Harmful. Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148 [[URL](https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf)]

# 

# Appendice A: tabella dei caratteri ASCII a 7 bit

Le seguenti tabelle contengono i 127 caratteri della codifica [ASCII](http://en.wikipedia.org/wiki/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** | | --- | --- | --- | --- | | 32 | 040 | 20 | *spazio* | | 33 | 041 | 21 | ! | | 34 | 042 | 22 | " | | 35 | 043 | 23 | # | | 36 | 044 | 24 | $ | | 37 | 045 | 25 | % | | 38 | 046 | 26 | & | | 39 | 047 | 27 | ' | | 40 | 050 | 28 | ( | | 41 | 051 | 29 | ) | | 42 | 052 | 2A | \* | | 43 | 053 | 2B | + | | 44 | 054 | 2C | , | | 45 | 055 | 2D | - | | 46 | 056 | 2E | . | | 47 | 057 | 2F | / | | 48 | 060 | 30 | 0 | | 49 | 061 | 31 | 1 | | 50 | 062 | 32 | 2 | | 51 | 063 | 33 | 3 | | 52 | 064 | 34 | 4 | | 53 | 065 | 35 | 5 | | 54 | 066 | 36 | 6 | | 55 | 067 | 37 | 7 | | 56 | 070 | 38 | 8 | | 57 | 071 | 39 | 9 | | 58 | 072 | 3A | : | | 59 | 073 | 3B | ; | | 60 | 074 | 3C | < | | 61 | 075 | 3D | = | | 62 | 076 | 3E | > | | 63 | 077 | 3F | ? | | 64 | 100 | 40 | @ | | 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 | | | **DEC** | **OCT** | **HEX** | **Simbolo** | | --- | --- | --- | --- | | 80 | 120 | 50 | P | | 81 | 121 | 51 | Q | | 82 | 122 | 52 | R | | 83 | 123 | 53 | S | | 84 | 124 | 54 | T | | 85 | 125 | 55 | U | | 86 | 126 | 56 | V | | 87 | 127 | 57 | W | | 88 | 130 | 58 | X | | 89 | 131 | 59 | Y | | 90 | 132 | 5A | Z | | 91 | 133 | 5B | [ | | 92 | 134 | 5C | \ | | 93 | 135 | 5D | ] | | 94 | 136 | 5E | ^ | | 95 | 137 | 5F | \_ | | 96 | 140 | 60 | ` | | 97 | 141 | 61 | a | | 98 | 142 | 62 | b | | 99 | 143 | 63 | c | | 100 | 144 | 64 | d | | 101 | 145 | 65 | e | | 102 | 146 | 66 | f | | 103 | 147 | 67 | g | | 104 | 150 | 68 | h | | 105 | 151 | 69 | i | | 106 | 152 | 6A | j | | 107 | 153 | 6B | k | | 108 | 154 | 6C | l | | 109 | 155 | 6D | m | | 110 | 156 | 6E | n | | 111 | 157 | 6F | o | | 112 | 160 | 70 | p | | 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: 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](http://it.wikipedia.org/wiki/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](#_304s4hj1jhh1) | interno | visualizza il percorso assoluto della directory corrente |
| [cd](#_pb1wxjuzqr0h) | interno | cambia directory corrente |
| [ls](#_jofw3djc1db1) | esterno | elenca il contenuto di una directory |
| [touch](#_110d7ncn2ehk) | esterno | crea un file vuoto o ne aggiorna la data di modifica |
| [mv](#_p5kj7u8sdp9w) | esterno | rinomina o sposta un file o una directory |
| [mkdir](#_47fq2gbcu0mq) | esterno | crea una nuova directory vuota |
| [rmdir](#_4t1fcpx7v1o8) | esterno | elimina una directory, purché sia vuota |
| [rm](#_nbbbywdqvk4e) | esterno | elimina un file o una directory |
| [cp](#_aijgsi5xitcx) | esterno | copia un file o una directory |

## B.1 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 corrisponde con 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. Questa ricerca viene effettuata solo in alcune specifiche directory del file system. Queste 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 fa matching con diversi binari (memorizzati all’interno di directory distinte), allora viene selezionata 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.

## B.2 Manipolazione ed esplorazione del file system

### B**.**2**.1** [pwd](http://www.freebsd.org/cgi/man.cgi?query=cd&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html): visualizza il percorso assoluto della directory corrente

| pwd |
| --- |
| Visualizza il percorso assoluto della directory corrente.  *Esempio:* $ pwd /home/studente/Scrivania |

### B.2.2 [cd](http://www.freebsd.org/cgi/man.cgi?query=cd&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html)**:** 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. |

### B.2**.3** [ls](http://www.freebsd.org/cgi/man.cgi?query=ls&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html)**: 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 directory specificata dal percorso (assoluto o relativo) nome-directory, 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 (rwx), e il suo contenuto è stato modificato il 27 settembre alle 13:16. |

### B.2**.4** [touch](http://www.freebsd.org/cgi/man.cgi?query=touch&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html)**:** 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. |

### B.2**.5** [mv](http://www.freebsd.org/cgi/man.cgi?query=mv&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html)**:** 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). |

### 

### B.2.6 [mkdir](http://www.freebsd.org/cgi/man.cgi?query=mkdir&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html): 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. |

### B.2.7 [rmdir](http://www.freebsd.org/cgi/man.cgi?query=mkdir&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html): elimina una directory, purché sia vuota

| rmdir directory |
| --- |
| Elimina la directory directory, purché sia vuota.  *Esempio 1:*  rmdir pippo  elimina la directory pippo dalla directory corrente.  *Esempio 2:*  rmdir /home/studente/Scrivania/pippo  elimina la directory pippo dalla directory /home/studente/Scrivania.  **Nota:** per eliminare directory non vuote, si veda il comando rm. |

### B.2.8 [rm](http://www.freebsd.org/cgi/man.cgi?query=rm&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html): 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 la directory pippo e tutto il suo contenuto dalla directory /home/studente/Scrivania. |

### B.2.9 [cp](http://www.freebsd.org/cgi/man.cgi?query=cp&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html): 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. |

## B.3 Altri comandi utili

| [geany](http://www.geany.org/) | 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 &  apre il file hello.c usando l'editor di testo geany. |
| --- | --- |
| [cat](http://www.freebsd.org/cgi/man.cgi?query=cat&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html) | Invia il contenuto di un file sul canale di output standard (di default è il terminale).  cat file: invia il contenuto del file file sul canale di output standard (di default è il terminale). |
| [less](http://www.freebsd.org/cgi/man.cgi?query=less&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html) | Visualizza nel terminale il contenuto di un file, consentendone di scorrere il contenuto con le frecce.  less file: visualizza il contenuto del file file nel terminale (premere q per uscire). |
| [man](http://www.freebsd.org/cgi/man.cgi?query=man&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html) | Visualizza nel terminale la documentazione di un comando esterno, di una libreria, di una system call, e di altri componenti del sistema.  man [section-id] comando: visualizza nel terminale la documentazione relativa al comando. section-id è un parametro opzionale che indica la categoria di documentazione che si vuole visualizzare. Valori tipicamente utili: 1 (Executable programs or shell commands), 2 (System calls (functions provided by the kernel), 3 (Library calls (functions within program libraries)). Se section-id viene omesso, man visualizza la documentazione trovata nella prima sezione (in ordine crescente) su cui avviene un match.  *Esempio 1:*  man cp  visualizza nel terminale la documentazione del comando cp.  *Esempio 2:*  man man  visualizza nel terminale la documentazione del comando man.  *Esempio 3:*  man 3 malloc  visualizza nel terminale la documentazione relativa alla funzione malloc della libreria C. |
| [env](http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_02.html) | Comando built-in che visualizza le variabili d’ambiente definite nella shell corrente.  *Esempio:*  $ env  SHELL=/bin/bash  USER=biar  PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin  [...] |
| [head](https://www.freebsd.org/cgi/man.cgi?query=head&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html) | Mostra le prime n righe di un file. Default n=10.  *Esempio 1:*  head file.txt  visualizza nel terminale le prime dieci righe di file.txt.  *Esempio 2:*  head -n 5 file.txt  visualizza nel terminale le prime 5 righe di file.txt. |
| [tail](https://www.freebsd.org/cgi/man.cgi?query=tail&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html) | Mostra le ultime n righe di un file. Default n=10.  *Esempio 1:*  tail file.txt  visualizza nel terminale le ultime dieci righe di file.txt.  *Esempio 2:*  tail -n 5 file.txt  visualizza nel terminale le ultime 5 righe di file.txt. |
| [which](https://www.freebsd.org/cgi/man.cgi?query=which&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html) | Localizza il binario che corrisponde a un comando esterno. Non gestisce correttamente comandi built-in o alias.  *Esempio 1:*  which cp  Viene restituito il percorso /bin/cp. |
| [whereis](https://www.freebsd.org/cgi/man.cgi?query=whereis&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html) | Localizza il binario che corrisponde a 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.  *Esempio 1:*  whereis cp  Viene restituito il percorso /bin/cp al binario ed il percorso /usr/share/man/man1/cp.1.gz alla sua documentazione. |
| [grep](https://www.freebsd.org/cgi/man.cgi?query=grep&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html) | 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.  *Esempio 1:*  grep parola file.txt  Solo le righe di testo in file.txt che contengono la stringa parola vengono emesse in output.  *Esempio 2:*  grep -v parola file.txt  Solo le righe di testo in file.txt che *non* contengono la stringa parola vengono emesse in output.  *Esempio 3:*  grep -n parola file.txt  Solo le righe di testo in file.txt che contengono la stringa parola vengono emesse in output. Per ogni riga emessa in output, viene riportato il numero di linea nel file originale. |
| [ln](https://www.freebsd.org/cgi/man.cgi?query=ln&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html) | Questo comando permette di creare dei link nel file system. Come trattato nel [Paragrafo 8.3](#_8lcm67cnhx8o), esistono due tipologie di link: *hard* link e *soft* link. Un soft link (anche detti link simbolico) offre una scorciatoia verso un determinato elemento del file system. Il comando ln permette di creare link simbolici quando usato con l’opzione -s.  *Esempio 1:*  ln -s /home/tom/Desktop/files/code /home/tom/files-with-code  Viene creato un link simbolico al percorso assoluto /home/tom/files-with-code che punta al percorso assoluto /home/tom/Desktop/files/code.  *Esempio 2:*  ln -s /home/tom/Desktop/files/text.txt text.txt  Viene creato un link simbolico con nome text.txt nella directory corrente che punta al percorso /home/tom/Desktop/files/text.txt. Analogamente si potrebbe anche far riferimento al percorso di destinazione del link con un percorso relativo. |

## B.4 Permessi Posix 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):
   1. “-”: file normale
   2. “l”: link simbolico (vedere comando ln)
   3. “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](https://www.freebsd.org/cgi/man.cgi?query=ls&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html), sezione *“The Long Format”*.

Come visto, i permessi di un file sono mostrati come una stringa di 9 caratteri (ad esempio **rwxrwxr-x**), che specifica i permessi per tre entità (**u**ser = utente proprietario, **g**roup = gruppo proprietario, **o**ther = 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 rwx ed un simbolo - indica l’assenza del permesso che occupa quella specifica posizione. La rappresentazione esplicita dei tre permessi rwx 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 rwx possono infatti essere mantenuti utilizzando tre bit b2b1b0, dove:

* b2 se settato ad 1 indica il permesso di lettura (r)
* b1 se settato ad 1 indica il permesso di scrittura (w)
* b0 se settato ad 1 indica il permesso di esecuzione (x)

Pertanto, si può esprimere la combinazione dei permessi assegnati a un'entità, calcolando in numero ottale, il valore ottenuto settando gli opportuni bit bi. Alcuni esempi (dove 2→8 indica il cambio di base da binario ad ottale) sono:

* 000 2→8 0: nessun permesso
* 001 2→8 1: permesso di esecuzione
* 010 2→8 2: permesso di scrittura
* 100 2→8 4: permesso di lettura
* 101 2→8 5: permesso di lettura ed esecuzione
* 110 2→8 6: permesso di lettura e scrittura
* 111 2→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 ed esecuzione per l’utente proprietario, il permesso di esecuzione per il gruppo proprietario, ed il permesso di lettura per i restanti utenti del sistema.

I permessi su di un file possono essere alterati attraverso il comando [chmod](https://www.freebsd.org/cgi/man.cgi?query=chmod&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html). 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](https://www.freebsd.org/cgi/man.cgi?query=chmod&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html). |

| 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 ed esecuzione 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](https://www.freebsd.org/cgi/man.cgi?query=chmod&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html). |

## B.5 Redirezione dello stdin, stdout, e stderr di un processo

La shell permette di redirigere i canali stdin, stdout e stderr per cambiare la loro destinazione. Vediamo alcuni esempi di redirezione dello stdout e stderr.

| cmd id > file |
| --- |
| Il canale di output identificato dal file descriptor id viene rediretto sul file file. Se viene omesso 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 omesso 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 rediretti 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. |

## B.6 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:  
   1. 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.
   2. 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 caso, 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 C: debugging di un programma 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*.

## C.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.

## C.2 Il debugger gdb

### C.2.1 Preliminari

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

| $ geany /home/studente/.gdbinit |
| --- |

Inserire nel file il seguente contenuto:

| define go  start  layout src  layout regs  focus cmd  end |
| --- |

### C.2.2 Compilare programma con i simboli di debugging

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

| gcc -m32 e\_main.c e.s -o e -g |
| --- |

Dove e.s contiene il nostro codice ASM ed e sarà l’eseguibile generato.

### C.2.3 Avviare il debugger

Per avviare la fase di debugging su un eseguibile e:

| $ gdb ./e |
| --- |

### 

### C.2.4 Comandi

#### C.2.4.1 Avvio e terminazione

| (gdb) go // esegue uno script che lancia il debugging del programma  // e si ferma sulla prima istruzione del main  (gdb) run [<arg1> ...] // per (ri)lanciare l'esecuzione con  // eventuali argomenti  (gdb) quit // per uscire da gdb (o ctrl-d) |
| --- |

#### C.2.4.2 Controllo esecuzione

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

#### C.2.4.3 Gestione breakpoint

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

#### C.2.4.4 Ispezione dello stato

| (gdb) print $eax // stampa il contenuto di un registro  (gdb) print x // stampa il contenuto della variabile x nello  // stack frame corrente  (gdb) x addr // stampa il contenuto della memoria all'indirizzo addr  (gdb) x $eax + 4 // stampa il contenuto della memoria all'indirizzo  // dato dall'espressione (eax + 4)  (gdb) x/nfu addr // stampa il contenuto della memoria all'indirizzo  // addr secondo il formato dal Paragrafo C.3.4.5 |
| --- |

#### C.2.4.5 Formato nfu

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

Per maggiori dettagli si veda la [documentazione](https://sourceware.org/gdb/onlinedocs/gdb/Output-Formats.html#Output-Formats).

#### C.2.4.6 Abbreviazioni dei comandi:

* print → p
* next → n
* cont → c
* step → s

#### C.2.4.7 Altri comandi

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

## C.3 Valgrind: debugging dell'uso della memoria

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***:*

| **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 unitialized-var.c -g -o unitialized-var | | --- |   Eseguiamo il programma:   | $ ./unitialized-var 0x8048461 | | --- |   L’output non sembra far emergere alcun problema nel programma. Nonostante ciò, se analizziamo l’esecuzione del programma sotto Valgrind:   | $ valgrind ./unitialized-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: ./unitialized-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 (unitialized-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 (unitialized-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 diversi 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 dallo 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:   | $ ./unitialized-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== 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).

# Appendice D: prerequisiti di elettronica

In questa appendice richiamiamo alcune nozioni di base di elettronica necessarie per comprendere i circuiti descritti nel [Paragrafo 1.1.2](#_xkozf0qbt8jk) e per illustrare possibili implementazioni di porte logiche.

## D.1 Resistenze e prima legge di Ohm

Uno dei circuiti più semplici che possono essere realizzati prevede quattro componenti: una **batteria**, un **interruttore**, una **resistenza** e un **diodo led** (o diodo a emissione luminosa) come mostrato in Figura D.1.1.

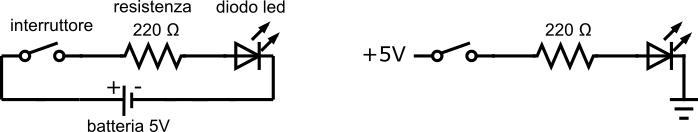


Figura D.1.1: semplice circuito per l'accensione di un diodo led.

L'obiettivo è semplicemente consentire all'interruttore di controllare l'accensione del led. Si noti che il circuito viene riportato con due convenzioni diverse: nella prima, la batteria appare esplicitamente. Nella seconda, il polo positivo viene riportato con il voltaggio e il polo negativo con il simbolo della massa (piramide rovesciata a trattini).

Un diodo led è un componente che agisce come dispositivo luminoso. Come tutti i diodi, ha due terminali: l'**anodo**, più lungo e connesso al polo positivo della batteria, e il **catodo**, più corto e connesso a quello negativo. Per limitare la corrente che attraversa il led in modo che non si danneggi, usiamo una resistenza che opera secondo la prima legge di Ohm:

| V = R⋅i |
| --- |

La corrente i si misura in Ampere, il voltaggio V della batteria in Volt, e la resistenza R in Ohm (anche indicata con la lettera greca 𝛀). La corrente di lavoro tipica di un led è compresa tra i 10 e i 30 Milliampere. Usando un'alimentazione a 5 Volt, in base alla legge di Ohm possiamo usare una resistenza compresa tra 5/0.030=166 e 5/0.010=500 Ohm.

Poiché sarebbe poco pratico per i produttori di componenti elettronici costruire resistenze per ogni possibile valore in Ohm, queste vengono prodotte in sottoinsiemi discreti di Ohm descritti da un codice espresso da bande colorate sulla resistenza come illustrato nella seguente tabella:

| **Colore** | **1° cifra** | **2° cifra** | **3° cifra** | **Moltiplicatore** | **Tolleranza** |
| --- | --- | --- | --- | --- | --- |
| Nero | 0 | 0 | 0 | 100 |  |
| Marrone | 1 | 1 | 1 | 101 | 1% |
| Rosso | 2 | 2 | 2 | 102 | 2% |
| Arancio | 3 | 3 | 3 | 103 | - |
| Giallo | 4 | 4 | 4 | 104 | - |
| Verde | 5 | 5 | 5 | 105 | 0.5% |
| Blu | 6 | 6 | 6 | 106 | 0.25% |
| Viola | 7 | 7 | 7 | 107 | 0.10% |
| Grigio | 8 | 8 | 8 | 108 | 0.05% |
| Bianco | 9 | 9 | 9 | 109 | - |
| Oro | - | - | - | 10-1 | 5% |
| Argento | - | - | - | 10-2 | 10% |

**Esempio 1**. La seguente resistenza a 4 bande colorate ha un valore di 0.2 Ohm con tolleranza 5%. Infatti, ha come bande: rosso=2 (1° cifra), nero=0 (2° cifra), argento=10-2 (moltiplicatore) e oro=5% (tolleranza)



Si ha quindi R = 20⋅10-2 = 0.2 Ohm.

**Esempio 2**. La seguente resistenza a 5 bande colorate ha un valore di 220 Ohm, e quindi può essere usata nel circuito illustrato precedentemente:



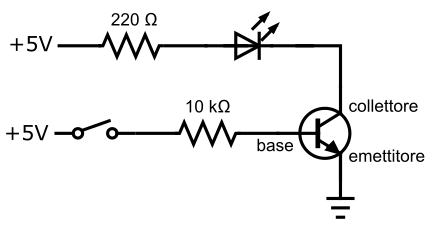
Infatti, ha come bande: rosso=2 (1° cifra), rosso=2 (2° cifra), nero=0 (3° cifra), nero=100 (moltiplicatore), marrone=1% (tolleranza). Si ha quindi R = 220⋅100 = 220 Ohm con tolleranza 1%.

## D.2 Transistor

I transistor sono dei componenti che hanno una duplice funzione: 1) possono agire come interruttori controllati non dalla pressione di un interruttore meccanico, ma dalla presenza di corrente su uno dei suoi terminali, e 2) possono amplificare un segnale elettrico. Nel nostro caso siamo interessati alla funzionalità 1), cruciale per la realizzazione di porte logiche.

In questa dispensa utilizzeremo transistor di tipo NPN che hanno tre connettori: **collettore**, **base** ed **emettitore**. Semplificando le cose, se il collettore è collegato al polo positivo e l'emettitore a massa, una piccola corrente applicata alla base fa fluire corrente tra il collettore e l'emettitore e il transistor si comporta come un interruttore chiuso. Se invece non vi è corrente applicata alla base, il transistor si comporta come un interruttore aperto e non vi è flusso di corrente tra emettitore e collettore.

Il seguente esempio illustra il comportamento sopra descritto, usando una resistenza da 10 K Ohm, vale a dire 5/10000 = 0.5 Milliampere sulla base del transistor:



Si noti che in questo semplice esempio la chiusura dell'interruttore provoca la chiusura del transistor e l'illuminazione del led.

## D.3 Breadboard

Una breadboard è una base universale che consente di realizzare circuiti anche complessi senza la necessità di saldare a stagno i componenti. Per questo motivo, è particolarmente indicata per la prototipazione di circuiti. In Figura D.3.1 mostriamo una breadboard classica che è caratterizzata da due file orizzontali su ciascun lato lungo. Una delle file porta il polo positivo (rosso) e l'altra il negativo (blu). Vi sono inoltre numerose file parallele al lato corto. I rettangoli neri evidenziano le file collegate tra loro elettricamente. Ogni fila ha dei piccoli fori in cui è possibile inserire le estremità di cavi elettrici (jumper) e i terminali dei componenti elettronici. La figura mostra anche una scheda Arduino, un semplice sistema di calcolo che nei nostri esempi verrà utilizzato per alimentare elettricamente a 5 Volt la breadboard prendendo corrente da una porta USB. In Figura D.3.2 mostriamo la realizzazione del circuito di Figura D.1.1.

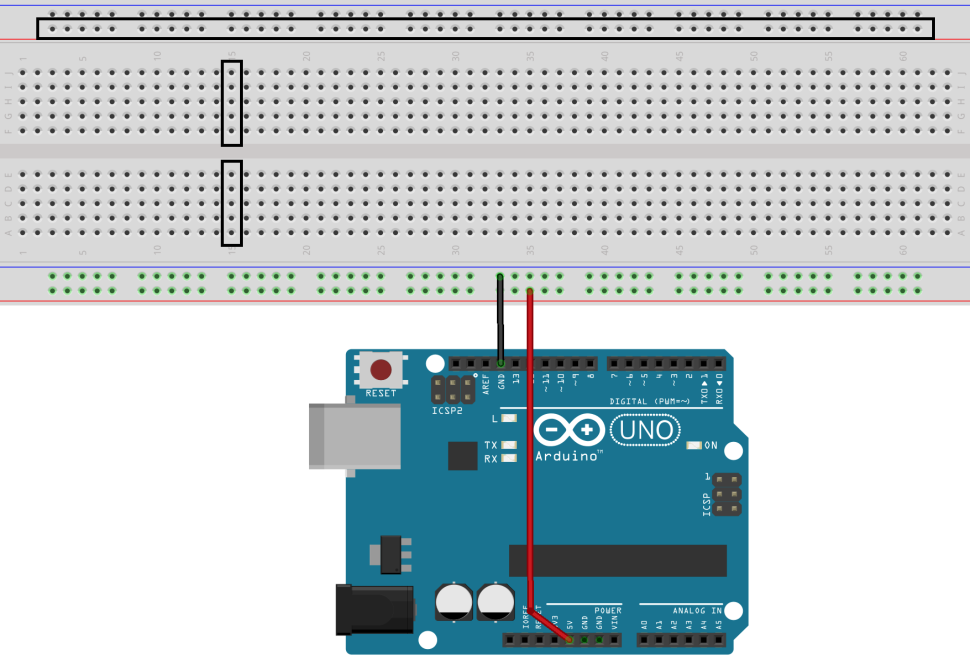


Figura D.3.1. Breadboard alimentata elettricamente da una scheda Arduino.



Figura D.3.1. Circuito di Figura D.1.1 realizzata mediante breadboard.

Vediamo ora una serie di esempi che ricadono in queste categorie e quali nformazioni Valgrindmette a disposizione dell’utente per identificarne le possibili cau

1. Esempi di famiglie di microprocessori sono [x86](http://en.wikipedia.org/wiki/X86) (Intel, AMD), [PowerPC](http://en.wikipedia.org/wiki/PowerPC) (Apple-Mototola-IBM), [SPARC](http://en.wikipedia.org/wiki/SPARC) (Sun Microsystems), [ARM](http://it.wikipedia.org/wiki/Architettura_ARM) (ARM Holdings). [↑](#footnote-ref-0)
2. Per semplicità in questa dispensa assumeremo che la frequenza di clock non vari nel tempo. Nei microprocessori moderni invece, la frequenza cambia a seconda delle circostanze per controllare la dissipazione del calore e il consumo energetico. [↑](#footnote-ref-1)
3. Come riferimento, un segnale elettrico richiede circa 10 nanosecondi per percorrere 15 centimetri lungo una connessione metallica [Balch 2003]. [↑](#footnote-ref-2)
4. Queste funzioni sono necessarie per rendere le system call, attivabili mediante un'istruzione assembly, invocabili come se fossero funzioni di libreria C. Si veda il paragrafo sulle [system call](#_nb1ikvyi7bzf). [↑](#footnote-ref-3)
5. Lo standard POSIX Edizione 2017 specifica il comportamento di oltre [400 funzioni](https://pubs.opengroup.org/onlinepubs/9699919799/functions/) fra cui le system call e le funzioni della libreria standard C (ISO C). [↑](#footnote-ref-4)
6. 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. [↑](#footnote-ref-5)
7. <http://en.wikipedia.org/wiki/Comparison_of_executable_file_formats> [↑](#footnote-ref-6)
8. <http://unix.stackexchange.com/questions/3322/what-makes-osx-programs-not-runnable-on-linux> [↑](#footnote-ref-7)
9. 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. [↑](#footnote-ref-8)
10. 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. [↑](#footnote-ref-9)
11. 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. [↑](#footnote-ref-10)
12. Si noti che in C il costrutto if…else può essere riscritto in termini di if e goto. [↑](#footnote-ref-11)
13. Si noti che in C il costrutto while può essere riscritto in termini di if e goto. [↑](#footnote-ref-12)
14. Si noti che in C il costrutto for può essere riscritto in termini di if e goto. [↑](#footnote-ref-13)
15. 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](https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/LowLevelABI/130-IA-32_Function_Calling_Conventions/IA32.html). [↑](#footnote-ref-14)
16. 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. [↑](#footnote-ref-15)
17. 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/mac/documentation/DeveloperTools/Conceptual/LowLevelABI/130-IA-32_Function_Calling_Conventions/IA32.html). [↑](#footnote-ref-16)
18. Per ottenere la documentazione completa sul comando man basta dare il comando man man. [↑](#footnote-ref-17)
19. 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. [↑](#footnote-ref-18)
20. <https://www.prowesscorp.com/computer-latency-at-a-human-scale/> [↑](#footnote-ref-19)
21. 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. [↑](#footnote-ref-20)
22. gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2 (gcc --version) [↑](#footnote-ref-21)
23. <https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/Optimize-Options.html#Optimize-Options> [↑](#footnote-ref-22)
24. 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. [↑](#footnote-ref-23)
25. 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. [↑](#footnote-ref-24)
26. Il comando grep estrae tutte le righe dell'output del comando ps -e che contengono la parola mem. [↑](#footnote-ref-25)
27. Si tratta di file che risiedono in memoria e possono essere acceduti come se fossero su disco. [↑](#footnote-ref-26)
28. r=lettura, w=scrittura, x=esecuzione, p=inaccessibile ad altri processi. [↑](#footnote-ref-27)
29. 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 bassa latenza. 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. [↑](#footnote-ref-28)
30. Interrupt Descriptor Table (IDT) nei sistemi x86. [↑](#footnote-ref-29)
31. 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. [↑](#footnote-ref-30)
32. Nei sistemi a 64 bit si usa un'istruzione assembly diversa e più efficiente (syscall) che non richiede l'accesso all'interrupt vector table. [↑](#footnote-ref-31)
33. 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. [↑](#footnote-ref-32)
34. E' possibile inviare segnali a gruppi di segnali, ma la trattazione va oltre gli scopi della dispensa. [↑](#footnote-ref-33)
35. <http://man7.org/linux/man-pages/man7/signal.7.html> [↑](#footnote-ref-34)
36. Numerazione vigente su sistemi x86, ARM e altri. [↑](#footnote-ref-35)
37. 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. [↑](#footnote-ref-36)
38. Per maggiori informazioni sulla sigaction si rimanda al sito [opengroup](https://pubs.opengroup.org/onlinepubs/9699919799/functions/sigaction.html%C3%B9%C3%B9). [↑](#footnote-ref-37)
39. <http://pubs.opengroup.org/onlinepubs/007904875/functions/alarm.html> [↑](#footnote-ref-38)
40. <http://pubs.opengroup.org/onlinepubs/007904875/functions/ualarm.html> [↑](#footnote-ref-39)
41. Standard ISO/IEC 80000 [↑](#footnote-ref-40)
42. Per semplicità, ci riferiamo al "blocco p" come al blocco il cui indirizzo iniziale è p. [↑](#footnote-ref-41)
43. Si pensi ad esempio a malware o virus. [↑](#footnote-ref-42)
44. Ignoriamo per semplicità calloc e realloc. [↑](#footnote-ref-43)
45. In alcuni ambienti la system call è deprecata a favore di mmap. [↑](#footnote-ref-44)
46. <https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html> [↑](#footnote-ref-45)
47. Si noti come un processo figlio creato da un genitore con una fork() possano condividere una stessa entry della tabella dei file aperti [↑](#footnote-ref-46)