Capitolo 1.

Il sistema operativo

> I'm not thinking about God as a super-smart intellect surrounded by large clusters of ultrafast Linux workstations and great search engines.
> That's the user's point of view.
>
> *(Donald E. Knuth — Things a computer scientist rarely talks about)*

1.1. Perché usiamo i sistemi operativi

1.1.1. La computazione

Un **algoritmo**[1] è la descrizione *priva di ambiguità* di un'attività di calcolo o, piú in generale, di una qualsiasi elaborazione di informazione. L'assenza di ambiguità è la caratteristica fondamentale che distingue un algoritmo da altri tipi di descrizione. L'esposizione di un algoritmo presuppone perciò la specifica dell'*interprete* respon-

[1] Per una trattazione dettagliata ma discorsiva del concetto di algoritmo si rimanda a: David Harel e Yishai Feldman. *Algoritmi. Lo spirito dell'informatica.* Trad. dall'inglese da Sam Guinea. Con pref. di Carlo Ghezzi. UNITEXT — Collana di informatica. Milano: Springer-Verlag Italia, 2008. pagg. xxii, 616. ISBN: 978-88-470-0579-2; Donald Ervin Knuth. *The art of computer programming: fundamental algorithms.* Inglese. III ed. Vol. 1. Addison-Wesley, 1997. pagg. xx, 650. ISBN: 0-201-89683-4

sabile della sua attuazione, o, in altre parole, di quali passi elementari, operazioni primitive o *istruzioni* sia possibile fare uso, mantenendo la certezza che l'interprete sia in grado di portarle a termine. Quando diamo indicazioni a un passante, descrivendo il modo con cui può raggiungere la meta, generalmente non enunciamo un vero e proprio algoritmo; in effetti di solito facciamo riferimento alla capacità di giudizio dell'ascoltatore, supponendola piú o meno analoga alla nostra. Nulla vieta però che la sua capacità di riconoscere "la casa rossa dopo la quale girare a destra" sia inficiata da daltonismo o scarsa illuminazione. Oppure potrebbe essere poco chiaro quando effettivamente svoltare: le strade senza uscita devono essere considerate?

L'informatica nasce dunque nel momento in cui si esplicita un interprete per l'elaborazione delle informazioni[2]. Gli interpreti con i quali hanno a che fare gli informatici che immaginano algoritmi possono essere astratti, come nel caso delle Macchine di Turing,[3] o concreti e reali dispositivi elettronici come un processore della famiglia Intel IA-32. In entrambi i casi è utile convenire delle regole sintattiche cui attenersi nello scrivere l'insieme di istruzioni che definiscono un dato algoritmo, che prenderà allora piú propriamente il nome di **programma**.

Programmare un algoritmo consiste dunque nel definirlo usando

[2]Naturalmente sono stati descritti "algoritmi" fin dall'antichità: il piú famoso dei quali è senz'altro quello di Euclide, di cui si tratta anche nel seguito; la parola stessa "algoritmo" deriva dal soprannome di un matematico arabo del IX secolo, Muḥammad ibn Mūsa, detto *al-Khuwārizmī* (cioé originario della Corasmia), che scrisse trattati in cui si spiegava come risolvere specifici problemi numerici, senza però chiarire quali abilità dovesse avere l'esecutore, un lettore "intelligente". La svolta fondamentale della formalizzazione di un interprete è frutto del lavoro sui fondamenti della matematica compiuti a cavallo fra il XIX e XX secolo, in particolare Alonzo Church e Alan M. Turing negli anni '30 definirono in maniera precisa cosa debba intendersi per "computabile".

[3]Charles Petzold. *The annotated Turing. A guided tour through Alan Turing's historic paper on computability and the Turing machine*. Inglese. Indianapolis, IN, USA: Wiley Publishing, Inc., 2008. pagg. xii, 372. ISBN: 978-0-470-22905-7.

i le operazioni elementari di cui l'interprete è capace. L'interprete, che svolge azioni puramente *meccaniche*[4], si dirà la *macchina* che esegue il programma. Per motivi di semplicità logica, efficacia costruttiva e/o economica, le macchine in questione potrebbero però fornire elaborazioni di base estremamente semplici o comunque caratterizzanti manipolazioni simboliche ben lontane da quelle concettualmente richieste per rendere facilmente comprensibile l'algoritmo (ossia espresso in termini simili a quelli che hanno aiutato a idearlo). Ci si aiuta perciò tramite notazioni compatte che vengono espanse in operazioni elementari *(macro-espansioni)* o progettando veri e propri linguaggi artificiali *(linguaggi di programmazione)* che — grazie alle loro proprietà formali — possono essere essi stessi algoritmicamente tradotti nel *linguaggio macchina.* Dal punto di vista del programmatore qualsiasi formalismo possa essere (magari dopo complicate e successive trasformazioni) ridotto all'interprete finale costituisce una vera e propria *macchina virtuale* in grado di eseguire attività computazionali.

1.1.2. Eseguire un programma

Un'attività computazionale nota in maniera semi-algoritmica fin dal V sec. a.C. è il cosiddetto *algoritmo di Euclide* usato per calcolare il Massimo Comun Divisore (MCD) di due interi positivi (vedi Programma 1.1).

Supponiamo di avere a disposizione un processore della famiglia IA-32, le cui caratteristiche sono riassunte nell'Appendice B. Possiamo descrivere il calcolo dell'MCD fra i numeri 420 e 240 con la notazione dell'assemblatore NASM (vedi Appendice C; la semantica delle istruzioni è comunque in buona parte descritta nei commenti

[4] Meccaniche nel senso che nessuna capacità di giudizio è necessaria alla corretta esecuzione; i fenomeni fisici coinvolti possono essere i piú disparati: meccanici, elettronici, ottici, ...

Programma 1.1: Algoritmo di Euclide

```
def mcd(x,y):
    assert (x > 0 and y > 0)
    if x == y: return x
    elif x > y: return mcd(x−y, y)
    else: return mcd(x,y−x)
```

(che seguono i ";") tramite uno pseudo-codice la cui sintassi è presa a prestito dal C, a sua volta descritto brevemente in Appendice D).

Programma 1.2: MCD di 420 e 240 in *assembly*

```
main:   mov dx, 420      ; dx = 420
        mov bx, 240      ; bx = 240
max:    cmp dx, bx       ; ZF = (dx == bx); CF = (dx < bx)
        je  fine         ; if (ZF) goto fine
        jg  diff         ; if !(ZF || CF) goto diff
        mov ax, dx       ; ax = dx
        mov dx, bx       ; dx = bx
        mov bx, ax       ; bx = ax
diff :  sub dx, bx       ; dx -= bx
        jmp max          ; goto max
fine :  hlt              ; halt /* dx == mcd(420,240) */
```

Per ragionare su cosa sia necessario fare perché questo programma possa essere eseguito possiamo anche considerare un modello assai semplificato dell'IA-32, ma che — per gli scopi che ci prefiggiamo ora — ne rappresenta una buona approssimazione: la cosiddetta Macchina di Von Neumann. In questo modello, rappresentato schematicamente in Figura 1.1, il dispositivo di calcolo è composto da un processore fornito di registri (che vengono usati per *accumulare* i risultati delle istruzioni), una memoria (accessibile come elenco ordinato di celle o parole di memoria) ed eventuali altre periferiche.

L'idea fondamentale dell'architettura di Von Neumann è che la memoria viene utilizzata per conservare sia il programma che i dati[5] sui quali opera (*stored program machine*) e lo *hardware* opera secondo il ciclo "*fetch*, *decode*, *execute*" per cui ad ogni iterazione (1) si recupera (caricandola nel registro istruzione corrente) un'istruzione dalla memoria, (2) si decodifica il significato dei bit e (3) si esegue l'istruzione ricavata dalla decodifica. Ciò permette di costruire macchine che hanno una parte "rigida", ossia identica in ogni realizzazione a prescindere dal problema che si intende risolvere, e una parte "flessibile", che occorre progettare ogni volta in modo che si adatti all'obiettivo computazionale che ci si propone. Ciò corrisponde alla consueta distinzione fra *hardware*, l'"attrezzatura" con cui si ottiene una volta per tutte il ciclo "*fetch*, *decode*, *execute*", e *software*, la "dottrina"[6] con cui si spiega come servirsene per ottenere un'elaborazione.

L'equivalente del programma 1.2 in linguaggio macchina è riportato in Tabella 1.1, dove la notazione esadecimale permette di sintetizzare quattro bit in ogni cifra. Dato che l'architettura IA-32 è *little-endian*[7] con parole di memoria da 32 bit (otto cifre esadecima-

[5] Questa è infatti la principale differenza con la macchina detta "di Harvard", che invece ha memorie distinte per dati e programmi; l'architettura di Harvard è molto utilizzata nei processori specializzati per l'elaborazione numerica dei segnali (*Digital Signal Processing*, DSP).

[6] *Software* è lemma il cui uso si è consolidato negli anni '60 del XX secolo (la prima occorrenza scritta nel senso odierno sembra essere in un articolo del 1958 "*The Teaching of Concrete Mathematics*" di John W. Tukey), voce coniata in opposizione all'aggettivo *hard*, duro, che è parte di *hardware*: in effetti un uso ottocentesco delle due parole si riferiva alla spazzatura, indicando con *software* la parte destinata alla rapida decomposizione e con *hardware* il resto. La voce *hardware* ha però il significato di utensíle e *hardware store* è, in inglese americano, il negozio della ferramenta (*iron monger* in inglese britannico). Il gioco di parole risulta impossibile in italiano: la proposta vorrebbe richiamarsi al gergo militare con cui si usa distinguere fra armamento e la sua dottrina d'impiego. In francese si parla di *matériel* e *logiciel*, una terminologia che attinge alla celebre distinzione cartesiana fra *res extensa* e *res cogitans*.

[7] Swift ne *I viaggi di Gulliver* chiama *Big-endian — Grosso-puntisti* in Jonathan

Figura 1.1.: La macchina di Von Neumann

li), occorre caricare in memoria (come si è detto, a partire dal primo bit di memoria, il n. 0) la sequenza di bit indicata in Tabella 1.2, dove gli indirizzi crescenti sono rappresentati dal basso verso l'alto, e il bit meno significativo è quello piú a destra, secondo la convenzione dei manuali Intel.

L'ambiente per operare efficacemente

Il programma potrebbe essere caricato in memoria in maniera *manuale,* agendo su appositi interruttori o tramite schede perforate: cosí in effetti avveniva nei primi calcolatori elettronici (vedi Figura 1.2). Volendola automatizzare — ossia volendola portare a termi-

Swift. *Viaggi di Gulliver in alcune remote regioni del mondo.* Trad. dall'inglese da Luigi de Marchi. V ed. Ulrico Hoepli Milano, 1943 — gli abitanti dell'isola di Blefuscu che ritengono che le uova vadano mangiate rompendone il guscio a partire dalla parte piú grossa. Nel gergo informatico con *little-endian* si indica un'architettura in cui i bit meno significativi di una parola di memoria hanno gli indirizzi piú piccoli, rappresentati in genere a partire da destra o dal basso. Molte architetture storicamente importanti (PDP-10, Motorola, IBM 370) erano *big-endian*, ossia usavano i primi bit per rappresentare le cifre piú significative (*biggest first*). Le sequenze di bit *big-endian* sono piú comode da leggere quando scritte orizzontalmente da sinistra a destra (corrisponde alla notazione ordinaria), mentre le sequenze *little-endian* sono piú adatte ad una lettura verticale.

Assembly	Linguaggio macchina
mov dx, 420	BAA401
mov bx, 240	BBF000
cmp dx, bx	39DA
je fine	740C
jg diff	7F06
mov ax, dx	89D0
mov dx, bx	89DA
mov bx, ax	89C3
sub dx, bx	29DA
jmp max	EBF0
hlt	F4

Tabella 1.1.: Traduzione delle istruzioni *assembly* in linguaggio macchina (in notazione esadecimale)

ne tramite un programma, il cosiddetto *loader* — si avranno sulla stessa macchina **due** programmi logicamente distinti: uno, infatti, *serve solo per poter eseguire convenientemente l'altro.*

Ecco allora la fondamentale distinzione fra *applicazioni* e *sistema operativo*: quest'ultimo è *software* che viene eseguito allo scopo di rendere *conveniente* la realizzazione e il funzionamento delle prime; serve cioè a creare *l'ambiente in cui le applicazioni operano.* In altre parole, acquistiamo lo *hardware* per poter usufruire delle applicazioni, le quali però funzionano in maniera efficace proprio grazie al sistema operativo. Una macchina priva di sistema operativo non vedrebbe certo diminuita la sua potenza ed espressività di calcolo, ma la realizzazione ed esecuzione dei programmi richiederebbe un cospicuo intervento umano, che viene invece considerevolmente ridotto quando l'ambiente operativo è ben progettato. Scopo del *software* che costituisce il sistema operativo è dunque in primo luogo semplificare al programmatore delle applicazioni la gestione

Indirizzo (byte)	Binario (76543210)	Esadecimale
0x00000016	11110100	F4
0x00000015	11110000	F0
0x00000014	11101011	EB
0x00000013	11011010	DA
0x00000012	00101001	29
0x00000011	11000011	C3
0x00000010	10001001	89
0x0000000f	11011010	DA
0x0000000e	10001001	89
0x0000000d	11010000	D0
0x0000000c	10001001	89
0x0000000b	00000110	06
0x0000000a	01111111	7F
0x00000009	00001100	0C
0x00000008	01110100	74
0x00000007	11011010	DA
0x00000006	00111001	39
0x00000005	00000000	00
0x00000004	11110000	F0
0x00000003	10111011	BB
0x00000002	00000001	01
0x00000001	10100100	A4
0x00000000	10111010	BA

Tabella 1.2.: La memoria dopo aver caricato il programma della Tabella 1.1 a partire (in basso) dall'indirizzo `0x00000000`; il bit meno significativo di ogni *byte* è il piú a destra e in *corsivo* è indicato l'inizio di una nuova parola di memoria.

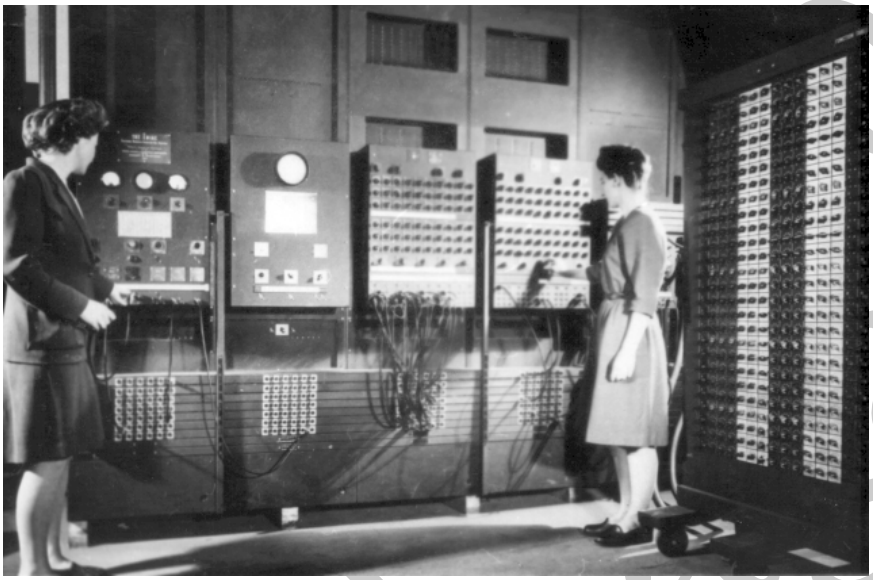

Figura 1.2.: Due donne programmano l'ENIAC (*US Army photo*, via Wikipedia)

dello *hardware*. Come vedremo, però, il suo ruolo non si esaurisce in questo: esso in definitiva fornisce ai programmatori moltissime astrazioni di alto livello che costituiscono la vera base con la quale realizzare le mirabolanti applicazioni cui siamo abituati.

Provare il programma 1.2 con QEMU

Grazie alla disponibilità di programmi capaci di emulare fin nei dettagli macchine IA-32 reali è possibile sperimentare abbastanza facilmente l'esecuzione di un programma *senza alcun sistema operativo*. Di seguito si mostrerà l'uso di QEMU [Fabrice Bellard. *QEMU, Open source processor emulator*. Inglese. URL: `http://wiki.qemu.org` (visitato il 31/12/2010)], ma analoghi esperimenti possono essere compiuti con altri emulatori di IA-32 di ampia diffusione (Bochs, VirtualBox, VMware, ...). QEMU mette a disposizione un *monitor* del sistema, ossia la possibilità di esaminare — *in*

corpore vili, per cosí dire — lo stato della macchina durante il suo funzionamento. Il *monitor* è accessibile tramite la combinazione di tasti `Ctrl-Alt-2`, a questo punto il comando `info registers` permette di visualizzare il contenuto di tutti i registri della macchina emulata (con `Ctrl-PgUp` e `Ctrl-PgDown` si può controllare ciò che è visibile sullo schermo); esistono inoltre anche comandi (`x` e `xp`) per esplorare il contenuto della memoria e controllare lo stato delle periferiche: una documentazione molto sintetica la si ottiene con il comando `help`. Il *monitor* è molto utile, ma per avere il massimo controllo è meglio far uso di un *debugger*, ossia di un programma che permette di fermare l'esecuzione nei punti critici (*breakpoint*) ed esaminare con calma lo stato della memoria, interpretandone il contenuto opportunamente. QEMU è predisposto per interagire con il *debugger* GDB [Free Software Foundation. *GDB: The GNU Project Debugger*. Inglese. URL: `http://www.gnu.org/software/gdb/` (visitato il 31/12/2010)] (per una brevissima introduzione ai comandi di GDB si veda l'Appendice F), che occorre mettere in comunicazione con QEMU tramite una opportuna connessione di rete.

Esecuzione della macchina QEMU senza alcun programma Si lanci l'emulatore, tenendo la macchina ferma (`-S`) e mettendo a disposizione l'interfaccia di *debugging* tramite una connessione di tipo TCP, in ascolto sulla porta 42000 (`-gdb tcp::42000`).

```
$ qemu -S -gdb tcp::42000
```

Parallelamente si lanci il *debugger* e lo si connetta all'emulatore. Se la connessione ha luogo correttamente, il *debugger* segnala di essere agganciato all'esecuzione di un programma *remoto*, che sta eseguendo l'istruzione all'indirizzo `0x0000fff0`.

```
$ gdb
(gdb) target remote localhost:42000
Remote debugging using localhost:42000
0x0000fff0 in ?? ()
```

A questo punto possiamo notare qualche dettaglio. Con il *debugger* è possibile osservare il contenuto della memoria: le prime 8 parole (da 32 bit) a partire dall'indirizzo `0x0` sono tutti zeri (anche 32 bit a zero sono un'istruzione IA-32 come è facile verificare variando il comando in `x/8i 0x0` in modo che l'interpretazione dei bit sia fatta secondo il codice macchina, anziché una semplice rappresentazione esadecimale).

```
(gdb) x/8x 0x0
0x0: 0x00000000 0x00000000 0x00000000 0x00000000
0x10: 0x00000000 0x00000000 0x00000000 0x00000000
```

Continuando con un esame sistematico della memoria, si scoprirebbe che è tutta piena di zeri fino all'indirizzo `0x0009ffff`, poi inizia una zona che i manuali Intel rivelano essere riservata ad usi speciali fino all'indirizzo `0x00100000`; da lí di nuovo zeri quasi fino alla fine dello spazio di indirizzamento a 32 bit (`0xffffffff`).

Quale istruzione sta per essere eseguita dalla macchina? Il *debugger* indica l'indirizzo istruzione corrente in `0x0000fff0`: all'accensione però il processore è in modalità reale; per garantire la compatibilità con i vecchi programmi, infatti, la modalità iniziale è simile a quella con cui funzionavano i processori Intel 8086. In particolare, gli indirizzi i (a 20 bit) in modalità reale vanno ottenuti sempre come somma di un indirizzo base b e uno spiazzamento o, secondo la formula $i = b \cdot 16 + o$, operazione piú facile di quanto sembri, visto che, in base 2, moltiplicare per 16 corrisponde ad un spostamento a sinistra di quattro posizioni, ossia una cifra esadecimale. In particolare l'indirizzo dell'istruzione corrente è composto dalla base da prelevare dal registro cs e lo spiazzamento da registro eip.

```
(gdb) info registers
eax            0x0 0
ecx            0x0 0
edx            0x633 1587
ebx            0x0 0
esp            0x0 0x0
ebp            0x0 0x0
esi            0x0 0
```

```
edi            0x0 0
eip            0xfff0 0xfff0
eflags         0x2 [ ]
cs             0xf000 61440
ss             0x0 0
ds             0x0 0
es             0x0 0
fs             0x0 0
gs             0x0 0
```

Il contenuto iniziale dei registri indica dunque che la prima istruzione da eseguire è all'indirizzo `0xf0000` + `0xfff0` = `0xffff0`. In altre parole, il costruttore (Intel) ha deciso che la prima istruzione che viene eseguita è quella che si trova all'indirizzo `0x000ffff0` (scritto piú propriamente `0xf0000:0xfff0` qualora si voglia rimarcare il fatto che sono significativi solo i 20 bit ottenuti secondo le regole dell'indirizzamento in modalità reale). Questa scelta è *cablata* nello *hardware* una volte per tutte, perciò per eseguire un programma qualsiasi occorrerebbe caricarlo a partire da tale indirizzo.

```
(gdb) set architecture i8086
The target architecture is assumed to be i8086
(gdb) x/i 0xffff0
   0xffff0: jmp    0xf000:0xe05b
```

All'indirizzo `0xf0000:0xfff0` c'è però già un'istruzione. Si tratta in effetti di un programma pre-caricato dal costruttore (e non facilmente modificabile), che è in grado a sua volta di caricare un programma dalla memoria di massa. In pratica un sistema operativo davvero minimale (di solito detto BIOS, *Basic Input/Output System*) che viene generalmente utilizzato per caricarne un sistema operativo piú completo (e scelto dall'utente, anziché dal costruttore). Obiettivo di questo primo esperimento è invece quello di sfruttare il BIOS per caricare il Programma 1.2: per riuscire occorre però attenersi ad alcune convenzioni. Per il momento si segua l'esecuzione del programma pre-caricato per un paio di istruzioni.

```
(gdb) si
0x0000e05b in ?? ()
(gdb) printf "Indirizzo istruzione corrente %x:%x\n", $cs, $eip
Indirizzo istruzione corrente f000:e05b
(gdb) x/i 0xfe05b
   0xfe05b: jmp    0xfc81e
(gdb) si
0x0000c81e in ?? ()
(gdb) printf "Indirizzo istruzione corrente %x:%x\n", $cs, $eip
Indirizzo istruzione corrente f000:c81e
(gdb) x/i 0xfc81e
   0xfc81e: mov    eax,cr0
(gdb) c
Continuing.
```

Il programma scrive sullo schermo (virtuale) alcuni messaggi d'errore, non essendo riuscito a caricare alcunché dalle memorie di massa.

Esecuzione del programma 1.2 Occorre preparare il Programma 1.2 in modo che possa essere caricato direttamente dal BIOS. Qualora lo si carichi da un disco magnetico (per i CD valgono regole meno restrittive), la convenzione da seguire è questa: il programma deve essere conservato nel primo settore del disco — che contiene al massimo 512 *byte* — e terminare (*byte* 511 e 512) con i *byte* AA55. Se queste condizioni sono soddisfatte, l'intera sequenza di 512 *byte* verrà caricata a partire dall'indirizzo `0x0000:0x7c00`. Una versione soddisfacente questi requisiti può essere ottenuta assemblando il Programma 1.3.

```
$ nasm -f bin -o mcdboot.bin mcd1.asm
$ qemu -hda mcdboot.bin -S -gdb tcp::42000
```

Il programma va assemblato in un formato contenente il puro codice macchina (`-f bin`). L'opzione `-hda mcdboot.bin` serve perché Qemu usi i *byte* contenuti nel *file* `mcdboot.bin` per emulare il primo *hard disk* (`-hda`).

```
(gdb) target remote localhost:42000
```

```
Remote debugging using localhost:42000
0x0000fff0 in ?? ()
(gdb) x/x 0x7c00
0x7c00: 0x00000000
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) set architecture i8086
The target architecture is assumed to be i8086
(gdb) x/11i 0x7c00
=> 0x7c00: mov    dx,0x1a4
   0x7c03: mov    bx,0xf0
   0x7c06: cmp    dx,bx
   0x7c08: je     0x7c16
   0x7c0a: jg     0x7c12
   0x7c0c: mov    ax,dx
   0x7c0e: mov    dx,bx
   0x7c10: mov    bx,ax
   0x7c12: sub    dx,bx
   0x7c14: jmp    0x7c06
   0x7c16: hlt
(gdb) si
0x00007c03 in ?? ()
(gdb) p $dx
$1 = 420
```

Il programma all'inizio non è in memoria, ma interrompendo (con un *breakpoint*) l'esecuzione al raggiungimento dell'indirizzo `0x0000:0x7c00`, si può controllare che è stato effettivamente caricato e seguirne l'evoluzione passo passo, osservando per esempio che nel registro dx è stato effettivamente caricato il valore 420.

Programma 1.3: MCD di 420 e 240 in *assembly*, adattato per essere caricato direttamente dal BIOS

```
segment .text
global  main

main:   mov dx, 420     ; dx = 420
        mov bx, 240     ; bx = 240
max:    cmp dx, bx      ; ZF = (dx == bx); CF = (dx < bx)
        je  fine        ; if (ZF) goto fine
        jg  diff        ; if !(ZF || CF) goto diff
        mov ax, dx      ; ax = dx
        mov dx, bx      ; dx = bx
        mov bx, ax      ; bx = ax
diff :  sub dx, bx      ; dx -= bx
        jmp max         ; goto max
fine :  hlt             ; halt /* dx == mcd(420,240) */

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

1.1.3. La gestione dello *hardware*

La gestione dello *hardware* non consiste naturalmente nel solo caricamento automatico dei programmi in memoria, ma soprattutto nel mascheramento della maggior parte dei dettagli legati alla tecnologia con cui la macchina è realizzata e, soprattutto, nel supporto alla gestione dello *hardware* periferico. In effetti, negli ambienti operativi moderni, la scrittura delle applicazioni può essere compiuta conoscendo ben poco dei dettagli delle singole periferiche: quali registri sono disponibili, quali tempistiche occorre rispettare, come affrontare i guasti temporanei, ecc. sono tutte problematiche di cui si deve occupare solo il programmatore di sistema nella realizzazione dei cosiddetti *device driver*, ossia le parti del sistema operativo deputate al controllo dell'*Input/Output* (I/O).

La scheda video

Torniamo ad esaminare il programma 1.2: l'unico modo per conoscere il risultato calcolato è esaminare il contenuto del registro dx quando l'esecuzione raggiunge l'indirizzo dell'ultima istruzione. Volendo mostrare il risultato in maniera piú conveniente per l'utente finale, occorre rendere disponibile il dato con una periferica adatta, per esempio lo schermo. Per far ciò occorre utilizzare la scheda video che controlla lo schermo, scrivendo i dati da visualizzare nella memoria della scheda, la memoria video. Se ci interessa usare la scheda in modalità testuale (ossia considerando lo schermo come una griglia rettangolare di caratteri), essa può essere generalmente pilotata secondo le convenzioni introdotte con l'IBM *Monochrome Display Adapter* (MDA), montata sul PC originale del 1981 (vedi Figura 1.3). L'MDA era dotata di 4KB di memoria, accessibili tramite indirizzi della memoria fisica del PC. Si tratta quindi della tecnica del cosiddetto *memory-mapped I/O*, che permette di accedere alle risorse dello *hardware* periferico tramite le consuete istruzioni di manipolazione della memoria di sistema: in pratica alcuni indiriz-

Figura 1.3.: IBM PC 5150 con tastiera e monitor monocromatico a fosfori verdi IBM 5151 (Foto di "Boffy b", via Wikipedia)

zi di memoria vengono fatti corrispondere a (in gergo, *mappati* su) parole di memoria gestite dalla periferica stessa. Nel caso dell'MDA la convenzione[8] consiste nel considerare gli indirizzi a partire da `0x0b0000` come corrispondenti ad un array di 80×25 = 2000 coppie di *byte* (un totale di 4 kB[9]), per 25 righe di 80 caratteri ciascuna: il primo *byte* serve a indicare il carattere da mostrare sullo schermo, mentre il secondo ne definisce alcuni parametri di visualizzazione (un'eventuale sottolineatura, l'intensità dell'illuminazione, la possibilità di lampeggiare, ecc.); ognuna delle 25 righe sullo schermo può quindi visualizzare fino a 80 caratteri.

La scelta del carattere originariamente avveniva indicizzando un elenco di 256 forme (*glyph*) cablate nella scheda video (si tratta della famosa *code page 437*, rappresentata in Figura 1.4): l'ordine

[8] Ralf Brown. *Ralf Brown's Interrupt List*. Inglese. Lug. 2000. URL: `http://www.cs.cmu.edu/~ralf/files.html` (visitato il 01/09/2009) (cit. come RBIL).

[9] 1 kB = 10^3 B = 1000 *byte*, vedi Appendice A.1

Figura 1.4.: Code page 437 (Immagine di "EatMyShortz", via Wikipedia)

dei caratteri di testo corrisponde a quello della codifica secondo l'*American Standard Code of Information Interchange* (ASCII), usata in NASM e in C (vedi Appendice A.2).

Memory-mapped I/O Convenzionalmente l'MDA (in modalità testuale) è mappata in memoria in modo tale che scrivendo un numero compreso fra 0 e 256 nelle celle di indirizzo pari a partire da `0xb8000` si ottiene la stampa a video del corrispondente caratteri ASCII. I dettagli cui il programmatore deve far caso sono però abbastanza numerosi. Per esempio:

- occorre impostare correttamente i parametri di visualizzazione per ogni carattere da stampare (tramite i *byte* di indirizzo dispari);
- occorre badare a non scrivere in zone dello schermo già occupate: le scritture in memoria, infatti, non considerano alcuna nozione di "cursore".

Accesso alla scheda video mappata in memoria

Scrivere nella parte di memoria in cui è mappata la scheda video è molto semplice, come mostra il Programma 1.4 che dovrebbe risultare di comprensione quasi immediata. Vale la pena notare appenda un paio di particolari:

1. l'indirizzo effettivo cui si fa riferimento nella prima **mov** del ciclo è calcolato secondo le regole della modalità reale: il primo indirizzo sarà pertanto `0xb800` $\cdot 16 + (100 - 2) =$ `0xb8062`, dopodiché si procede a ritroso, saltando il *byte* relativo ai parametri di visualizzazione;
2. l'indice del ciclo **cx** è automaticamente decrementato dall'istruzione **loop**.

Il supporto del BIOS per la gestione del video Il *software* di base pre-caricato potrebbe fornire una **libreria**, ossia una biblioteca[10] di funzionalità utili a semplificare la scrittura dei programmi ed evitare le ripetizioni di codice. In effetti è proprio cosí: nei PC IBM compatibili la zona di memoria da `0xA0000` a `0x100000` è in sola lettura[11] (*Read-Only Memory*, ROMROM) e in essa sono pre-caricate una serie di *routine* per la gestione dello *hardware*: si tratta del cosiddetto *firmware* — *software* fornito dal produttore e cablato nella memoria della macchina durante la costruzione — come il classico *Basic Input/Output System* (BIOSBIOS).

Come si accede alle *routine* del *firmware*? Si potrebbe pensare di farlo tramite una chiamata di procedura (**call**). Il produttore del *firmware* dovrebbe pubblicare gli indirizzi ai quali sono caricate le

[10] In effetti questo è il vero significato della parola inglese *library*. Ma la parola italiana è molto *flessibile* e significa anche "scaffale adibito alla conservazione di libri" (*bookshelf*) e non solo "negozio di libri" (*bookshop*).

[11] In realtà a volte è possibile modificarla con operazioni particolari, in gergo si parla di fare il *flash* del *firmware*.

Programma 1.4: Scrittura di caratteri sullo schermo tramite *memory-mapped I/O*

```
segment .text
global  main

QUANTI equ 100                  ; #define QUANTI 100
N equ QUANTI*2-2                ; #define N (QUANTI*2-2)
main:
        mov ax, 0xb800          ; ax = 0xb800 /* mov ds, x è vietato */
        mov ds, ax              ; ds = ax /* ds == 0xb800 */
        mov cx, QUANTI          ; cx = QUANTI /* cx indice loop */
        mov bx, N               ; bx = N
ciclo :                         ; do {
        mov byte [ds:bx], 'm'   ; mem[ds:bx] = 'm'
        sub bx, 2               ; bx -= 2
        loop ciclo              ; } while (cx != 0);
fine :
        hlt

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

routine e versioni diverse potrebbero richiedere indirizzi differenti, creando non pochi problemi di compatibilità con le versioni precedenti. Si preferisce allora utilizzare una tecnica di *chiamata implicita* che, come vedremo nel § 1.2, ha anche il vantaggio di permettere l'utilizzo di meccanismi *hardware* di protezione ed è per questo motivo uno dei meccanismi fondamentali utilizzato nel dialogo fra applicazioni e sistema operativo. La chiamata implicita sfrutta il meccanismo *hardware* delle richieste di interruzione, attivato però tramite un'istruzione che prende perciò il nome di interruzione *software*. L'effetto è del tutto analogo a quello di un'interruzione generata da una componente *hardware* : il processore si occupa di salvare automaticamente parte del proprio stato e salta all'indirizzo di un apposito gestore dell'interruzione che può essere impostato programmando opportunamente il controllore delle interruzioni e la memoria; al termine viene poi ripristinato lo stato antecedente l'interruzione stessa. Nell'architettura IA-32 un'interruzione *software* può essere ottenuta con un'istruzione (int) del programma in esecuzione, in maniera del tutto prevedibile e *sincrona.* Il produttore del *firmware* — generalmente il primo programma che viene eseguito all'accensione della macchina — non dovrà far altro che inizializzare la tabella dei gestori delle interruzioni con i dati necessari a raggiungere le *routine* di gestione dello *hardware* e pubblicare l'associazione fra una data interruzione e la periferica controllata (per IA-32 si veda *Ralf Brown's Interrupt List*[12]). Si noti che le interruzioni *software* vengono utilizzate in questo caso per dare una struttura logica coerente e conveniente alla comunicazione fra CPU e periferiche quando l'iniziativa parte dalla CPU stessa. Nel caso delle interruzioni *hardware,* invece, l'iniziativa della comunicazione parte dalle periferiche e si parlerà piú propriamente di richiesta d'interruzione|seeInterrupt Request (*Interrupt Request*, IRQ).

Nel caso del BIOS dei PC IBM compatibili l'interruzione 16 (`0x10`) viene utilizzata per attivare una complessa *routine* di gestione della

[12] RBIL

scheda video, in grado di evidenziare anche un cursore per segnalare la posizione "corrente". Il protocollo secondo il quale deve essere (implicitamente) chiamata prevede che si usi il *byte* basso del registro **ax** (**al**) per comunicare il carattere da stampare e il *byte* alto (**ah**) per stabilire la modalità di visualizzazione (`0x0e` indica per esempio la modalità testuale standard); inoltre i bit nel registro **bx** stabiliscono gli attributi "estetici" come il colore e la luminosità. La scrittura di un carattere può quindi sostanzialmente ridursi al seguente semplice frammento:

```
mov al, 'x'    ; al = 'x' /* stampa 'x' in modalita' testuale */
mov ah, 0x0E   ; ah = 0x0e /* BIOS: scrivi sullo schermo (mod. std) */
mov bx, 0      ; bx = 0 /* BIOS: bh page number, bl foreground */
int 0x10       ; /* BIOS: scrivi al sullo schermo e sposta il cursore */
```

Il programma 1.5 incapsula la stampa in una funzione put e, per semplicità, stampa il risultato del calcolo con una rappresentazione binaria identica a quella usata internamente dalla macchina.

Salto esplicito alla *routine* di gestione dello *hardware*

È possibile sostituire un salto esplicito alla *routine* di gestione dello *hardware* alla chiamata implicita tramite interruzione? Per sperimentare usiamo una versione del Programma 1.4 in cui la scrittura avviene tramite il BIOS, con la procedura put (Programma 1.6). Per prima cosa si identifichi l'indirizzo del gestore dell'interruzione.

```
$ nasm -f bin hello-int.asm
$ qemu -hda hello-int -S -gdb tcp::42000 &
$ gdb
(gdb) target remote localhost:42000
Remote debugging using localhost:42000
0x0000fff0 in ?? ()
(gdb) set architecture i8086
The target architecture is assumed to be i8086
(gdb) b *0x7c00
```

Programma 1.5: MCD di 420 e 240 in *assembly*, con output via BIOS

```
main:   mov dx, 420       ; dx = 420
        mov bx, 240       ; bx = 240
max:    cmp dx, bx        ; ZF = (dx == bx); CF = (dx < bx)
        je  fine          ; if (ZF) goto fine
        jg  diff          ; if !(ZF || CF) goto diff
        mov ax, dx        ; ax = dx
        mov dx, bx        ; dx = bx
        mov bx, ax        ; bx = ax
diff :  sub dx, bx        ; dx -= bx
        jmp max           ; goto max
fine :  nop               ; no operation /* dx == mcd(420,240) */

        mov cx, 16        ; cx = 16 /* contatore loop (implicito) */
sbit :  shl dx, 1         ; dx «= 1
        jnc zero          ; if (CF == 0) goto zero

        mov al, '1'       ; al = '1'
        call put          ; /* stampa 1 */
        jmp ciclo         ; goto ciclo

zero:   mov al, '0'       ; al = '0'
        call put          ; /* stampa 0 */

ciclo : loop sbit         ; if (cx > 0) { cx -= 1; goto sbit; }
        mov al, 10        ; al = 10 /* ASCII next line */
        call put          ; /* stampa il carattere 'next line' */
        mov al, 13        ; al = 13 /* ASCII carriage return */
        call put          ; /* stampa il carattere 'carriage return' */
stop:   hlt               ; ferma il processore

put:                      ; /* stampa al in modalità testuale */
        mov ah, 0x0E      ; ah = 0x0e /* BIOS: write char to screen */
        mov bx, 0         ; bx = 0 /* BIOS: bh page num bl foreground */
        int 0x10          ; /* BIOS: write al to screen */
        ret               ; return
```

```
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/10i 0x7c00
=> 0x7c00: mov    ax,0xb800
   0x7c03: mov    ds,ax
   0x7c05: mov    cx,0x64
   0x7c08: mov    al,0x6d
   0x7c0a: call   0x7c10
   0x7c0d: loop   0x7c08
   0x7c0f: hlt
   0x7c10: mov    ah,0xe
   0x7c12: mov    bx,0x0
   0x7c15: int    0x10
(gdb) b *0x7c15
Breakpoint 2 at 0x7c15
(gdb) c
Continuing.

Breakpoint 2, 0x00007c15 in ?? ()
(gdb) si
0x00008424 in ?? ()
(gdb) p/x $cs
$4 = 0xc000
```

Ora si può provare a sostituire l'istruzione di **int** 0x10 con un **jmp** 0xc000:0x8424. Funziona, ma viene stampato un solo carattere perché il programmatore del BIOS non ha (giustamente) previsto alcuna istruzione di ritorno al codice chiamante, e anche una **call** non funzionerebbe meglio: l'istruzione **iret** non avrà l'effetto desiderato, dato che l'alterazione del flusso di controllo è stata ottenuta con l'istruzione **jmp**.

Programma 1.6: Scrittura di caratteri sullo schermo tramite *routine* BIOS

```
segment .text
global  main

QUANTI equ 100              ; #define QUANTI 100
N equ QUANTI*2-2            ; #define N (QUANTI*2-2)
main:
        mov ax, 0xb800      ; ax = 0xb800 /* mov ds, n è vietato */
        mov ds, ax          ; ds = ax /* ds == 0xb800 */
        mov cx, QUANTI      ; cx = quanti /* cx indice loop */
ciclo :                     ; do {
        mov al, 'm'         ; al = 'm' /* BIOS: char to print */
        call  put           ; put()
        loop  ciclo         ; } while (cx != 0);
fine :
        hlt
put:                        ; /* stampa al in modalità testuale */
        mov ah, 0x0E        ; ah = 0x0e /* BIOS: write char to screen */
        mov bx, 0           ; bx = 0 /* BIOS: bh page bl foreground */
        int  0x10           ; /* BIOS: write al to screen */
        ret                 ; return

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

La tastiera

Il programma 1.5 è ancora molto primitivo. Una limitazione particolarmente fastidiosa è che il calcolo dell'MCD fra due dati differenti da 420 e 240 richiede necessariamente una modifica al programma stesso e una conseguente ri-compilazione e caricamento in memoria. Per ottenere i dati dall'utente del programma bisogna coinvolgere nel funzionamento una periferica di input, per esempio la tastiera. Anche qui esiste una scheda di controllo (*controller*) di riferimento: l'i8042. Anche l'i8042 potrebbe essere gestito in maniera analoga alla scheda video, agendo su registri mappati in memoria[13] oppure sfruttando i servizi predefiniti messi a disposizione del BIOS. L'occasione è però propizia per presentare una tecnica alternativa: l'uso delle *porte di I/O*.

Porte di I/O Molti processori sono in grado di utilizzare *due* spazi di indirizzamento separati e alternativi (vi si accede tramite istruzioni dedicate): il primo — e generalmente molto piú ampio — è riservato alla memoria, mentre il secondo è utilizzato per alcune periferiche, tipicamente quelle che hanno registri in cui si conservano pochi *byte*. Nell'architettura IA-32 quest'ultimo consiste in 65536 indirizzi chiamati porte di I/O (*I/O port* in inglese[14]), cui si accede tramite le istruzioni **in** (per trasferire dati dalla periferica al registro **ax**) e **out** (per trasferire dati dal registro **ax** alla periferica). Tipicamente le schede di controllo delle periferiche hanno diversi registri a ciascuno dei quali viene fatta corrispondere una porta: (1) un registro di stato, che permette di riconoscere se la periferica è pronta ad operare; (2) uno o piú registri dati; (3) un registro di controllo, per inviare alla periferica comandi specifici.

[13] L'indirizzo convenzionale nei PC IBM compatibili è `0x0080:0x0040`. Per maggiori dettagli si rimanda a RBIL

[14] Il significato principale della parola *port* è "porto" e solo secondariamente "porta" (di una città): in effetti anche le zone deputate a ospitare le porte *hardware* prendono il nome di *bay*, "baia".

Nel caso dell'i8042, una lettura della porta `0x0064` corrisponde ad una lettura del registro di stato, mentre la porta `0x0060` è associata al registro dati del *controller*. L'interpretazione dei bit del registro dati è piuttosto articolata (vedi[15]) e qui basti menzionare che il bit 0 indica se il buffer di *output* (ossia quello in cui la tastiera accumula i suoi dati) è pieno. La rilevazione della pressione o rilascio di un tasto può quindi effettuarsi cosí:

```
read:
  in  al, 0x64  ; al = get_from_port(0x0064)
  test al, 1b   ; ZF = ( al & 0x1 )
  jz  read      ; if (ZF) goto read
  in  al, 0x60  ; al = get_from_port(0x0060)
```

***Busy waiting* e interruzioni** Lo schema presentato è un classico esempio della cosiddetta *busy waiting*. Il processore, infatti, ripete un ciclo di lettura fino a quando non si verifica una determinata condizione: si tratta quindi di un'*attesa operosa*, in cui il processore esegue istruzioni con l'unico scopo di aspettare lo scorrere del tempo necessario al verificarsi dell'evento di interesse. In effetti, prima che l'utente prema (o rilasci) un tasto potrebbero passare tempi anche molto lunghi (almeno dal punto di vista della velocità del processore) e in questo intervallo la CPU lavora a pieno regime, senza che sia possibile, per esempio, attivare alcuna modalità di risparmio energetico. Ecco perché in caso del genere risulta molto piú efficiente affidarsi a una soluzione basata sulle *interruzioni hardware*. La periferica deve essere in grado di attivare una linea di interruzione e il gestore di tale interruzione sarà incaricato di rilevare il dato d'interesse. Poiché il gestore viene eseguito solo quando viene attivata la richiesta di interruzione, il processore non viene caricato di operazioni inutili. Il codice va un po' complicato, però, perché occorre registrare il gestore dell'interruzione (il 9, nel caso della tastiera)

[15] RBIL.

nella tabella delle interruzioni (i dettagli necessari alla gestione di questa tabella cambiano quando il processore funziona in modalità reale o protetta, vedi Appendice B); inoltre il gestore dovrà anche preoccuparsi di comunicare l'avvenuta gestione scrivendo opportunamente sulla porta `0x0020` che comanda il *Programmable Interrupt Controller* (PIC).

```
;; /* registrazione gestore */
    cli                          ; /* disabilita interruzioni */
    mov ax, 0                    ; ax = 0
    mov ds, ax                   ; ds = ax
    mov word [ds :(9*4)], kbd    ; mem[ds:9*4] = kbd /* offset, gestore 9 */
    mov word [ds :(9*4)+2], cs   ; mem[ds:9*4+2] = cs /* segmento, gestore
9 */
    sti                          ; /* riattiva interruzioni */
;; /* gestore */
kbd:                             ; /* gestore interruzioni */
    pusha                        ; /* salva lo stato dei registri (stack) */
    in   al, 0x60                ; al = get_from_port(0x0060)
    mov [x], al                  ; x = al
    mov al, 0x20                 ; al = 0x20
    out 0x20, al                 ; put_to_port(0x0020, al) /* PIC ack
*/
    popa                         ; /* ripristina registri dallo stack */
    iret                         ; /* return from interrupt */
;; /* dati */
x:  db 0                         ; char x = 0
```

mcd3

MCD completo

1.1.4. La multi-programmazione

Il diagramma a) della Figura 1.5 mostra l'evoluzione temporale esecuzione del programma che gestisce l'*input* da tastiera tramite

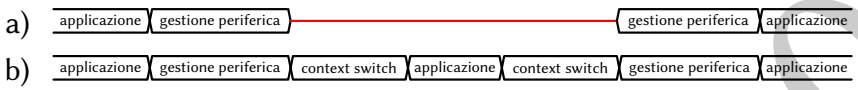

Figura 1.5.: Evoluzione nel tempo dell'attività della CPU in sistemi a) uni-programmati e b) multi-programmati.

interruzioni. Come si vede, c'è un periodo piuttosto lungo in cui la CPU non deve eseguire alcuna istruzione e può quindi essere posta in uno stato di semi-attività a consumo ridotto di potenza (*idle state*). In realtà in questo periodo "d'ozio" il processore sarebbe a disposizione per eseguire istruzioni che non dipendano dall'operazione di I/O in corso. La tecnica della multi-programmazione consiste nel gestire in memoria piú di un'applicazione per volta, in modo che sia possibile passare all'esecuzione di una o dell'altra (operando il cosiddetto cambiamento di contesto esecutivo, *context switch*) secondo l'opportunità, seguendo ingegnose politiche di pianificazione (*scheduling*) per ottimizzare l'utilizzo dello *hardware* a disposizione. Il diagramma b) della Figura 1.5 mostra questa idea: occorre aggiungere della computazione di servizio (*overhead*) per effettuare il cambio di contesto (operazione che va fatta in maniera tale da potere poi ripristinare esattamente lo stato precedente), ma in questo modo è possibile sfruttare pienamente la CPU.

Il sistema operativo deve quindi gestire la nozione di *applicazione in esecuzione* che prende tradizionalmente il nome di **processo**.

Dal punto di vista dell'utilizzatore del sistema, la gestione dei processi produce una parallelizzazione delle attività anche in presenza di un unico processore (pseudo-parallelismo). Infatti, si considerino due attività A_1 e A_2 che isolatamente impiegherebbero rispettivamente i tempi t_1 e t_2, comprensivi dei periodi di inattività della CPU dovuti a operazioni di I/O; l'utente di un sistema (uni-processore) con una oculata gestione dei processi vedrà il termine di entrambe le attività dopo un tempo $t < t_1 + t_2$, come se A_1 e A_2 fossero state eseguite (parzialmente) in parallelo.

Dal punto di vista del programmatore, invece, la multi-programmazione crea l'illusione di avere un processore dedicato ad ogni processo. Infatti, il salvataggio e conseguente ripristino dello stato del processore durante i cambi di contesto, crea una situazione equivalente a quella che si avrebbe se ogni programma in esecuzione avesse a disposizione registri dedicati: in pratica è come se esistessero n copie di tutti i registri del processore, per n processi in esecuzione pseudo-parallela.

L'utilità della tecnica dello pseudo-parallelismo guidato dalle interruzioni hardware è tale che in molti sistemi si introduce un'interruzione periodica (*clock*) che permette di variare il processo in esecuzione decine di volte in un secondo anche in assenza di attività di I/O e molta dell'attività del sistema operativo è volta ad ottenere in modo efficiente che l'utente abbia la possibilità di portare avanti molte attività in maniera concorrente (*multitasking*).

1.2. La "cipolla"

Il sistema operativo potrebbe a questo punto sembrare sostanzialmente una ben fornita biblioteca di funzionalità dalla quale i programmatori delle applicazioni pescano i servizi di cui hanno bisogno per realizzare i loro obiettivi computazionali con maggiore facilità. Questo è senz'altro un modo di vedere le cose e talvolta per il programmatore non è neanche facile distinguere i servizi del sistema operativo da quelli forniti dalle librerie di base, come la `libc`, la libreria con la quale sono generalmente collegati i programmi scritti in C: infatti anche le *routine* che richiedono l'intervento del sistema operativo sono generalmente incapsulate in librerie, il cui uso permette di aumentare la portabilità dei programmi. Dal punto di vista implementativo c'è però da notare che se la differenza fra i servizi forniti dal sistema operativo e altre librerie fosse solo nominale, ciò significherebbe che il programmatore delle librerie avrebbe la stessa libertà d'azione del programmatore del sistema operativo.

Per confinare l'impatto degli inevitabili errori, risulta invece assai utile porre un discrimine ben preciso fra quello che può fare l'uno e l'altro. Questo discrimine è ottenuto con un supporto *hardware*: i processori piú diffusi funzionano in modalità diverse quando eseguono codice che fa parte del sistema operativo piuttosto che quello delle librerie applicative. È abitudine consolidata rappresentare lo schema logico che ne deriva con la Figura 1.6, che ricorda la sezione di una cipolla. Nel caso più semplice gli strati (i *ring*, come vengono chiamati attingendo dalla tradizione gastronomica americana degli *onion ring*, anelli di cipolla) sono due (sistema operativo e applicazioni), oltre all'inevitabile *hardware*, ma è possibile fare uso di stratificazioni piú complesse se il processore ne è capace. Nel caso dell'IA-32 le modalità di funzionamento sono 4, ma i sistemi che ne sfruttano piú di 2 sono abbastanza rari (in MINIX, p. es., se ne usano 3). La modalità in cui è permessa qualsiasi operazione (compreso il cambiamento di modalità!) è chiamata *kernel-mode* o *supervisor-mode* perché è appunto il modo in cui opera il kernel ossia il nucleo essenziale del sistema operativo: quella parte che proprio perché è l'unica che viene eseguita senza restrizioni deve essere sempre in memoria ed eseguibile immediatamente. Viceversa la modalità in cui alcune istruzioni critiche sono vietate prende il nome di modo utente (*user-mode*).

1.2.1. Chiamate di sistema

Naturalmente è possibile utilizzare le istruzioni eseguite in modo *kernel* per cambiare il livello di privilegio ed eseguire (da lí in poi) istruzioni in modo utente. Il contrario invece necessita particolari cautele. In effetti se semplicemente esistesse un'istruzione non privilegiata di cambio di modo, sarebbe possibile eseguire istruzioni privilegiate in un programma applicativo (vedi Tabella 1.3(a)): il che è proprio ciò che si vuole evitare. Si ricorre pertanto a cambi di modalità *impliciti* tramite istruzioni che hanno l'ulteriore effetto di

Figura 1.6.: Il modello a "cipolla" dei sistemi di calcolo

spostare il flusso di controllo a indirizzi che contengono istruzioni di sistema (vedi Tabella 1.3(b)).

La tecnica con la quale in modo utente si richiede l'esecuzione dei servizi del *kernel* è perciò basata sulla *chiamata implicita*, e ciò avviene con l'attivazione di un'interruzione *software*, con una procedura analoga a quanto descritto nel § 1.1.3 per le interruzioni *hardware*: la differenza è che in questo caso l'interruzione è del tutto prevedibile e *sincrona* visto che è stata introdotta dal programmatore. L'interruzione *software* usata a questo scopo è chiamata anche *trap*, proprio perché l'esecuzione finisce per essere *intrappolata* nello spazio di memoria allocato al *kernel*, il cui codice deciderà come gestire la prosecuzione. Questo meccanismo delle *trap* è fondamentale e si ripete ogni volta che parti con privilegi differenti devono interagire: il codice privilegiato può chiamare direttamente il subordinato, ma l'operazione inversa non è possibile e bisogna ricorrere ad una chiamata implicita. Vediamo di chiarire ulteriormente il concetto con una metafora: s'immagini di avere una classe di studenti, uno dei quali si rende conto che nell'aula c'è una bomba pronta ad

programma	istruzione	modalità
...	...	...
applicazione	ordinaria	*user*
applicazione	ordinaria	*user*
applicazione	cambia modo	*user*
applicazione	**privilegiata**	*kernel*

(a) Situazione da evitare

programma	istruzione	modalità
...	...	...
applicazione	ordinaria	*user*
applicazione	ordinaria	*user*
applicazione	*trap*	*user*
sistema	**privilegiata**	*kernel*

(b) Lo *hardware* garantisce che in modo *kernel* si eseguano istruzioni di sistema

Tabella 1.3.: Lo *hardware* garantisce che dopo un cambio di modo **non possano** essere eseguite istruzioni dell'applicazione (a), ma si salti a istruzioni di sistema tramite una "chiamata implicita" (b)

esplodere; poichè è assolutamente vietato che gli studenti disinneschino le bombe (l'assicurazione universitaria non copre tali rischi), lo studente scrive sulla lavagna il motivo di pericolo e l'ubicazione della bomba e, dopo aver azionato l'allarme, lascia l'aula insieme ai suoi compagni; sentito l'allarme, il responsabile della sicurezza arriva nell'aula, trova l'indicazione del problema sulla lavagna e decide di disinnescare la bomba, dopo di che può richiamare gli studenti, che continueranno le loro attività.

Chiamate di sistema dirette

La richiesta di un servizio del sistema operativo si riduce quindi a caricare nei registri (o in memoria) i dati opportuni e sollevare l'interruzione *software*. Allo scopo è possibile che *hardware* metta a disposizione specifiche istruzioni di *trap* (l'IA-32 dispone dell'istruzione `sysenter` p. es.) oppure si sfrutta la generazione di un'interruzione (il cui numero è una convenzione interna al sistema operativo) come per le chiamate implicite del § 1.1.3. Il listato 1.7 mostra il codice utente equivalente alla chiamata di libreria `exit` con il parametro attuale 42: in effetti la `libc` potrebbe contenere codice analogo. Infatti, vale la pena ribadire che anche il codice delle librerie di base (p. es. la `libc`) gira in modalità utente. Quando viene sollevata l'interruzione `0x80`, però, entra in gioco il *kernel* — le cui istruzioni vengono quindi eseguite in modo *kernel* — che, nel caso di Linux convenzionalmente identifica il servizio richiesto leggendo il registro **eax** e, per la `exit`, l'unico parametro dal registro **ebx**. Ogni servizio è identificato da un numero, secondo la tabella delle chiamate di sistema, gestita dal sistema operativo. Per usare i servizi bisogna quindi conoscerne il numero (reperibile in `<sys/syscall.h>`), anche se generalmente ciò non è necessario grazie all'intermediazione delle librerie di base.

Programma 1.7: Codice (in modo utente) che richiede exit (42) in Linux

```
section .text
global main

main:
 mov eax, 1    ; system call n.1: exit
 mov ebx, 42   ; parametro della system call: exit(42)
 int 0x80      ; trap 128, secondo la convenzione Linux
```

1.2.2. *Kernel* monolitici e *micro-kernel*

Non tutto il codice del sistema operativo gira in modo kernel.

monolitico – micro-kernel chiamate di sistema tramite messaggi

1.3. Astrazioni

Il modello a "cipolla" trasforma il sistema operativo in un fornitore di astrazioni per il livello applicativo. I programmatori di quest'ultimo, infatti, possono vedere soltanto uno *hardware* virtuale, come appare attraverso il filtro delle *chiamate di sistema.*

Memoria virtuale La piú fondamentale di queste astrazioni è senza dubbio la *memoria virtuale*: cioè l'astrazione secondo cui i programmi utente vedono la memoria *hardware*. Generalmente al programmatore viene data l'illusione di avere a disposizione uno spazio di indirizzamento piatto e interamente sotto il proprio controllo: per le macchine a 32 bit, potrebbe essere una sequenza di 4 Gi di celle di memoria, magari da un B, e questo a prescindere dal fatto che il medesimo *hardware* ospiti altri programmi (e il sistema operativo!). Le idee fondamentali per la gestione della memoria verranno trattate nel § 5.

Programmi in esecuzione Come abbiamo visto i sistemi operativi nascono proprio come strumenti di supporto all'esecuzione di programmi applicativi. Allo scopo il sistema fornisce l'astrazione chiave di *programma in esecuzione*. Tradizionalmente questa astrazione si realizza tramite il concetto di **processo**, una struttura dati gestita dal sistema operativo che permette di mantenere le informazioni necessarie all'avvio, all'eventuale interruzione e alla ripresa dell'esecuzione di un programma, garantendo un isolamento pressoché totale rispetto ad altre attività concorrenti per le medesime risorse *hardware*. I sistemi operativi moderni forniscono talvolta altre astrazioni per l'esecuzione di sequenze di istruzioni, con caratteristiche e garanzie differenti: i *thread*, p. es. possono eseguire con memoria condivisa e cessione al programmatore applicativo di parte della responsabilità della sincronizzazione. Queste astrazioni verranno discusse nel § 3.

Persistenza dei dati La memoria centrale è generalmente realizzata con tecnologie elettroniche di tipo *volatile*, capaci cioè di conservare un dato solo durante il periodo in cui sono alimentate dalla corrente elettrica. Da qui discende la necessità di avere anche una *memoria secondaria* in cui conservare dati in maniera persistente: in particolare le istruzioni del sistema operativo e delle applicazioni che devono essere caricate all'avviamento dello *hardware*.

Il concetto di *persistenza* non è solo legato alle limitazione della tecnologia usate per realizzare gli elementi di memoria, però: infatti un'astrazione importante è la possibilità di considerare dati che *persistono* al termine dell'esecuzione di un programma e possono essere perciò utilizzati in un'esecuzione successiva o da un altro programma. Tradizionalmente i sistemi UNIX hanno realizzato questa astrazione tramite il concetto di *file*, sequenze di *byte* che vengono catalogati in maniera gerarchica in *directory* (che a loro volta possono essere considerate *file*). Il concetto di *file* è estremamente povero, ma proprio per questo molto flessibile: UNIX utilizza

infatti astrazioni simili per l'accesso alle periferiche, la comunicazioni fra processi e di rete. Un'analisi piú dettagliata del concetto di *file* e di come implementare un *file system* si trova nel § 7.

Interazione col sistema operativo Naturalmente l'utente di un sistema deve interagire con la macchina virtuale realizzata dal sistema operativo. Ci sono due casi d'uso (*use case*) che vale la pena distinguere perché danno luogo a interazioni molto differenti:

1. l'utente che intende *programmare* la macchina virtuale realizzata dal sistema operativo: in questo caso l'interazione avverrà secondo le modalità imposte dalle chiamate di sistema, che costituiscono una API (*Application Programming Interface*) non eludibile da alcuna istruzione in *user-mode*. Questa tipologia d'interazione è discussa nel § 2, con riferimento allo standard POSIX.

2. l'utente che intende servirsi del sistema operativo per eseguire applicazioni, manipolando e condividendo dati fra di esse. Anche in questo caso l'utente dovrà indicare/descrivere ciò che vuole, ma tipicamente lo fa con modalità specializzate (p. es. un interprete di comandi testuali molto simile a un linguaggio di programmazione tradizionale, oppure un'interfaccia *point-and-click*) che meritano una trattazione a sè. Come è ovvio, è proprio in questa modalità di interazione che si ritrova la maggiore varietà fra i sistemi operativi: il confine dell'interfaccia è tutto sommato piuttosto lasco e nulla vieta di modificarlo, tipicamente agendo nella modalità di interazione precedente, cioè creando nuovi programmi applicativi! L'interazione tramite interprete di comandi testuali è discussa nel § 4, con riferimento alla *shell* POSIX.

1.4. Riassumendo

Il *software* che costituisce il sistema operativo ha quindi due funzioni fondamentali:

1. fornire all'utente servizi che permettono l'esecuzione dei programmi applicativi e le altre attività a ciò correlate (come il salvataggio dei dati); per l'utente il sistema operativo è soprattutto il **gestore dei programmi applicativi**.

2. fornire al programmatore meccanismi che permettono lo sviluppo delle applicazioni prescindendo dalla contemporanea esecuzioni di altri programmi sconosciuti; per il programmatore il sistema operativo è il **gestore delle risorse del sistema**.

3. fornire al programmatore astrazioni che permetto lo sviluppo delle applicazioni in maniera efficace e sicura; per il programmatore il sistema operativo è quindi soprattutto un fondamentale **fornitore di astrazioni**.

Capitolo 2.

L'interfaccia del programmatore: le chiamate di sistema POSIX

> An x64 processor is screaming along at billions of cycles per second to run the XNU kernel, which is frantically working through all the POSIX-specified abstraction to create the Darwin system underlying OS X, which in turn is straining itself to run Firefox and its Gecko renderer, which creates a Flash object which renders dozens of video frames every second.
>
> Because I wanted to see a cat jump into a box and fall over.
>
> I am God.
>
> *(Randall Munroe — Abstraction*
> `http://xkcd.com/676/`*)*

Per il programmatore delle applicazioni l'interfaccia del sistema operativo sono le chiamate di sistema[1] (vedi § 1.2.1), che quindi co-

[1] A rigore questo è vero soltanto per il programmatore in linguaggio *assembly* o macchina: linguaggi di piú alto livello nascondono dietro all'intermediazione del