Introduzione ai sistemi operativi

Daniel P. Bovet

corso di “Sistemi Operativi”

Facoltà di Ingegneria
Università di Roma “Tor Vergata”

21 dicembre 2010
# Indice

## 1 PROGRAMMAZIONE DI SISTEMI

| 1.1 | Introduzione | 3 |
| 1.2 | Le librerie di programmi | 6 |
| 1.3 | Realizzazione di funzioni di libreria: il ruolo delle API | 9 |
| 1.4 | La programmazione di sistemi: una anteprima | 10 |

## 2 EVOLUZIONE DEI SISTEMI OPERATIVI

| 2.1 | Introduzione | 11 |
| 2.2 | Software di base e sistemi operativi | 11 |
| 2.3 | Classificazione dei sistemi operativi | 13 |
| 2.3.1 | Sistemi transazionali | 13 |
| 2.3.2 | Sistemi di controllo | 14 |
| 2.4 | Evoluzione storica | 15 |
| 2.4.1 | Sistemi dedicati (prima versione) | 16 |
| 2.4.2 | Gestione a lotti | 16 |
| 2.4.3 | Sistemi interattivi | 19 |
| 2.4.4 | Sistemi dedicati (seconda versione) | 21 |
| 2.5 | Virtualizzatori | 21 |
2.6 Sviluppi futuri ............................................. 22

3 SOFTWARE DI BASE ........................................... 23
  3.1 I componenti di un sistema operativo .................. 23
  3.2 Messa a punto di programmi ............................. 24
    3.2.1 Assemblatori, compilatori e interpreti ............ 24
  3.3 Linkaggio di programmi .................................. 26
    3.3.1 Linkaggio statico ................................... 28
    3.3.2 Linkaggio dinamico ................................ 30
    3.3.3 Librerie statiche e dinamiche ...................... 32
    3.3.4 Immissione di programmi ............................ 34
    3.3.5 Caricamento in memoria di un file eseguibile ...... 34
  3.4 Supporto durante l’esecuzione .......................... 34
  3.5 Salvataggio/ripristino di dati .......................... 36
  3.6 Amministrazione del sistema ............................ 37

4 IL FILE SYSTEM ................................................. 39
  4.1 Introduzione ............................................. 39
  4.2 Caratteristiche dei file system ........................ 40
  4.3 Protezione delle informazioni .......................... 41
  4.4 Il filesystem di Unix ................................. 41
    4.4.1 File ............................................... 41
    4.4.2 Link hard e soft ................................... 43
    4.4.3 Tipi di file ....................................... 45
    4.4.4 Descrittore di file ed inode ....................... 46
## INDICE

| 4.4.5 | Diritti d’accesso e file mode | 47 |
| 4.4.6 | API per la gestione di file | 48 |
| 4.4.7 | API POSIX per la gestione di directory | 49 |
| 4.4.8 | Chiamate di sistema per la gestione di file | 49 |
| 4.4.9 | Apertura di un file | 50 |
| 4.4.10 | Indirizzamento di un file | 51 |
| 4.4.11 | Chiusura di un file | 52 |
| 4.4.12 | Cambiamento di nome e cancellazione di file | 53 |

### 5 INTERFACCIA CON L’UTENTE

| 5.1 | Introduzione | 55 |
| 5.2 | Interfaccia a menu | 55 |
| 5.3 | Interfaccia a comandi | 56 |
| 5.3.1 | Lo shell di Unix | 57 |
| 5.3.2 | Un esempio di script bash | 59 |
| 5.3.3 | Struttura dello shell | 59 |
| 5.4 | Interfacce grafiche | 61 |
| 5.4.1 | X Window | 62 |
| 5.5 | Realizzazione di interfacce grafiche in Unix | 64 |

### 6 NUCLEO E PROCESSI

<p>| 6.1 | Introduzione | 65 |
| 6.2 | Architettura e modello di esecuzione | 67 |
| 6.3 | Processi | 68 |
| 6.4 | Il NUCLEO coordinatore di processi | 70 |</p>
<table>
<thead>
<tr>
<th>INDICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5 Risorse usate dai processi</td>
</tr>
<tr>
<td>6.6 Scheduling di processi</td>
</tr>
<tr>
<td>6.6.1 Prevedibilità dei tempi d’esecuzione</td>
</tr>
<tr>
<td>6.6.2 Utilizzazione delle risorse</td>
</tr>
<tr>
<td>6.6.3 Scheduling a breve termine</td>
</tr>
<tr>
<td>6.6.4 Scheduling a lungo termine</td>
</tr>
<tr>
<td>7 PROGRAMMAZIONE CONCORRENTE</td>
</tr>
<tr>
<td>7.1 Introduzione</td>
</tr>
<tr>
<td>7.2 Schemi di sincronizzazione</td>
</tr>
<tr>
<td>7.2.1 Attesa</td>
</tr>
<tr>
<td>7.2.2 Mutua esclusione</td>
</tr>
<tr>
<td>7.2.3 Produttore/consumatore</td>
</tr>
<tr>
<td>7.3 Primitive di sincronizzazione</td>
</tr>
<tr>
<td>7.4 Come realizzare un programma concorrente</td>
</tr>
<tr>
<td>7.5 Supporto Unix</td>
</tr>
<tr>
<td>7.5.1 Creazione/eliminazione di processi</td>
</tr>
<tr>
<td>7.5.2 Segnali</td>
</tr>
<tr>
<td>7.5.3 Una alternativa ai semafori delle API IPC</td>
</tr>
<tr>
<td>7.6 Stallo tra processi</td>
</tr>
<tr>
<td>7.7 Programmazione in tempo reale</td>
</tr>
<tr>
<td>8 SPAZIO DEGLI INDIRIZZI DI UN PROCESSO</td>
</tr>
<tr>
<td>8.1 Introduzione</td>
</tr>
<tr>
<td>8.2 Spazio degli indirizzi di un processo</td>
</tr>
<tr>
<td>Capitolo</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>8.3</td>
</tr>
<tr>
<td>8.4</td>
</tr>
<tr>
<td>8.5</td>
</tr>
<tr>
<td>8.5.1</td>
</tr>
<tr>
<td>8.5.2</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>9.1</td>
</tr>
<tr>
<td>9.2</td>
</tr>
<tr>
<td>9.3</td>
</tr>
<tr>
<td>9.4</td>
</tr>
<tr>
<td>9.5</td>
</tr>
<tr>
<td>9.6</td>
</tr>
<tr>
<td>9.7</td>
</tr>
<tr>
<td>9.8</td>
</tr>
<tr>
<td>9.9</td>
</tr>
<tr>
<td>9.10</td>
</tr>
<tr>
<td>9.11</td>
</tr>
<tr>
<td>9.12</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10.1</td>
</tr>
<tr>
<td>10.2</td>
</tr>
<tr>
<td>10.2.1</td>
</tr>
<tr>
<td>10.2.2</td>
</tr>
</tbody>
</table>
INDICE

10.3 L'algoritmo Buddy System .................................. 126
10.4 Estensioni del Buddy System ................................. 128
10.5 Demand Paging ................................................. 129
10.6 Recupero di memoria libera .................................. 131
10.7 Gestione dello spazio su disco ............................... 133

11 GESTIONE DEI DISPOSITIVI DI I/O .......................... 135

11.1 Introduzione .................................................. 135
11.2 Architettura di I/O ............................................. 135
11.3 Dispositivi di I/O riconosciuti dal file system .............. 138
   11.3.1 Programmazione di un device file ...................... 138
11.4 Supporto del NUCLEO ....................................... 140
11.5 Sincronizzazione tra CPU e dispositivo di I/O ............. 141
11.6 Uso di software cache nei driver per dischi ............... 142
Capitolo 1

PROGRAMMAZIONE DI SISTEMI

1.1 Introduzione

Prima di immergerci nello studio dei sistemi operativi, cerchiamo di convincere il lettore che vi sono validi motivi per affrontare tale studio. La nostra tesi è che esistono vari tipi di programmazione e che, soltanto conoscendo bene i sistemi operativi, risulta possibile scrivere programmi di un certo tipo.

In effetti, oltre alla programmazione più semplice che si impara in poche settimane, esistono altri tipi di programmazione che richiedono anni di duro impegno per poterli dominare correttamente.

Questo fatto è evidenziato nel mondo produttivo da apposite qualificazioni professionali: il “programmatore” è nella gerarchia aziendale qualcosa di assai diverso dal “programmatore di sistemi”.

Per i nostri scopi, sorvoleremo sul fatto che esistono vari tipi di programmazione e distingueremo soltanto quattro livelli livelli:

1. indipendente dall’hardware e dal sistema operativo
2. dipendente dal sistema operativo ma indipendente dall’hardware
3. dipendente dal sistema operativo e dall’hardware
CAPITOLO 1. PROGRAMMAZIONE DI SISTEMI

4. dipendente dai chip installati sulla scheda madre

Vediamo in cosa differiscono tali livelli.

Programmi indipendenti dall’hardware e dal sistema operativo

Il primo livello è ovviamente quello che garantisce la massima portabilità dei programmi tra piattaforme hardware e software di tipo diverso. Le specifiche di molti linguaggi ad alto livello quali Cobol, FORTRAN o C sono state rigidamente definite da appositi comitati di standardizzazione. In conseguenza, risulta possibile scrivere programmi che possono essere compilati ed eseguiti correttamente su calcolatori di tipo diverso e/o su calcolatori che usano sistemi operativi di tipo diverso.

Altri linguaggi di programmazione più recenti quali C++ o Java sono stati anch’essi standardizzati ma ciò non è sufficiente ad assicurare la totale portabilità. In effetti, tali linguaggi sono in qualche modo ”estensibili” e consentono al programmatore di avvalersi di classi predefinite le quali non sono state standardizzate. Nel caso di Java, ad esempio, esistono diversi livelli di “Java virtual machine” ognuna delle quali riconosce un particolare sottoinsieme di classi.

Vedremo nella sezione 1.2 come un problema simile si verifica col linguaggio C quando il programmatore fa uso di librerie non standard.

Programmi dipendenti dal sistema operativo ma indipendenti dall’hardware

La versatilità dei linguaggi di programmazione sviluppati negli ultimi anni ha un costo: essi non consentono di accedere direttamente ad informazioni gestite dal sistema operativo bensì attraverso apposite costrutti del linguaggio o, nel caso di linguaggi orientati ad oggetto, tramite apposite classi. Ciò viene fatto di proposito per rendere il linguaggio di programmazione indipendente dal sistema operativo utilizzato.

Vi sono tuttavia casi in cui il programmatore intende ottimizzare le prestazioni del suo programma sfruttando appieno le funzionalità del sistema operativo. Per fare ciò, il programmatore fa uso di apposite funzioni di interfaccia offerte dal sistema operativo. Vedremo più avanti in questo capitolo come ciò avviene. A questo punto, vogliamo soltanto evidenziare come questo tipo di programmazione chiamata programmazione di sistema consente di sviluppare programmi più
1.1. INTRODUZIONE

sofisticati a scapito però della portabilità: un programma di sistema potrà essere ricompilato ed eseguito correttamente su macchine diverse purché esse usino tutte lo stesso sistema operativo usato per realizzare il programma (Windows XP, Linux, AIX, ecc.).

Programmi dipendenti dal sistema operativo e dall’hardware

Esiste un terzo livello di programmazione riservato a pochi specialisti a cui è necessario ricorrere quando la programmazione di sistema non è sufficiente. Ciò avviene quando si richiede di scrivere un nuovo sistema operativo, oppure quando si richiede di ampliare le funzionalità di un sistema esistente.

In tale caso, ovviamente, non è possibile avvalersi di apposite funzioni di interfaccia ma, al contrario, viene richiesto al programmatore di realizzarne delle nuove.

Un caso classico è quello della realizzazione dei cosiddetti driver di I/O per nuove periferiche. Quando viene immessa sul mercato una nuova apparecchiatura periferica (scheda video, sound card, ecc.), è necessario aggiungere ai driver esistenti un nuovo driver per la periferica in modo che essa possa essere “riconosciuta” dal sistema operativo.

Come vedremo alla fine del corso, i driver di I/O sono dei programmi inclusi nel sistema operativo che interagiscono con la scheda hardware di interfaccia del dispositivo. In questo senso, sono programmi aventi un grado di portabilità praticamente nullo, in quanto possono essere eseguiti soltanto su piattaforme hardware dello stesso tipo che fanno uso dello stesso sistema operativo e, addirittura, di una specifica versione di uno specifico sistema operativo.\footnote{Come vedremo più avanti, i “nuclei” o “kernel” dei sistemi operativi evolvono costantemente e le funzioni offerte per realizzare driver di I/O possono cambiare, per cui un driver realizzato per una “vecchia” versione del kernel può risultare incompatibile con un kernel più recente.}

Programmi dipendenti dai chip presenti nella scheda madre

Tali programmi sono eseguiti subito dopo l’accensione del sistema e sono spesso indicati con il nome di BIOS (Basic Input/Output System). Essi eseguono in modalità Real Address Mode. La memoria indirizzabile è limitata al primo megarbyte. Nella prima fase del POST (Power On Self Test), la RAM non è utilizzabile perché i controllori dei chip della RAM non sono stati ancora inizializzati ed il
BIOS fa uso della cache L2 del processore. Il BIOS è generalmente codificato in Assembly a 16 bit anche se sono recentemente avviati alcuni progetti open source, ad esempio Coreboot, che usano il C anziché l’Assembly a 16 bit. Tali programmi non dipendono da alcun sistema operativo, dipendono invece dai chip presenti nella scheda madre. Ogni scheda madre ha un proprio BIOS registrato in una memoria non volatilé. Al momento dell’accensione della macchina la CPU passa ad eseguire la prima istruzione del BIOS.

1.2 Le librerie di programmi

Ogni linguaggio di programmazione offre la possibilità di segmentare il programma in sottoprogrammi più semplici. Nel linguaggio C, il programma è composto da una o più funzioni\(^2\). La funzione principale, quella invocata per prima quando il programma viene eseguito, è chiamata main(). Altre funzioni possono essere introdotte a piacere dal programmatore.

Per semplificare il lavoro del programmatore, i produttori di software hanno messo a punto collezioni di funzioni di uso comune raggruppate in librerie\(^3\) specializzate. Nel caso di linguaggi orientati ad oggetti, si parlerà di classi ma il concetto è lo stesso. Poiché il nostro linguaggio di riferimento è il C, useremo il termine libreria.

Dal punto di vista della portabilità dei programmi, è necessario distinguere due tipi di librerie:

- librerie standardizzate dal linguaggio di programmazione
- altre librerie non standardizzate offerte dal sistema operativo oppure definite dal programmatore.

La differenza tra i due tipi di librerie è cruciale dal punto di vista della portabilità dei programmi: un programma mantiene la portabilità tra sistemi basati su piattaforme hardware e software diverse se fa uso di funzioni appartenenti a librerie del primo tipo. Vice versa, non è più garantita la portabilità dei programmi che fanno uso di funzioni incluse in librerie del secondo tipo.

\(^2\)Il termine funzione indica che ognuna di esse, ad eccezione di quelle di tipo void, restituisce un qualche valore, ad esempio, un intero, un puntatore ad una struttura, ecc.

\(^3\)Il termine librerie è un anglicismo entrato nel vocabolario comune. In italiano, sarebbe più corretto parlare di biblioteche.
1.2. LE LIBRERIE DI PROGRAMMI

Per semplicità, introdurremo la nozione di libreria di programmi facendo riferimento al linguaggio Cermo restando che ogni linguaggio di programmazione include le proprie librerie di programmi.

**Librerie standardizzate**

Nel linguaggio C, per potere fare uso di una funzione standard, ossia ad una funzione appartenente ad una qualche libreria standardizzata, è necessario premettere nel programma una apposita frase di tipo:

```c
#include <xxx.h>
```

ossia una direttiva di inclusione di un *header file* che descrive i prototipi di tutte le funzioni incluse nella libreria (o nelle librerie) associate all’identificatore xxx. Se, ad esempio, si intende usare la ben nota funzione fwrite() per scrivere caratteri di testo in un file, è necessario premettere nel programma la frase:

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

descrive i prototipi di tutte le funzioni dello “stream I/O”.

Si noti come l’uso delle `#include` lascia all’implementatore del sistema operativo totale libertà riguardo al modo di realizzare le librerie. Quello che viene standardizzato è il prototipo della funzione e il nome dello header file. In Linux, lo header file `stdio.h` è contenuto nel file `/usr/include/stdio.h`4. Se apriamo tale file tramite un editor di testo, ritroviamo in corrispondenza alla fwrite() una dichiarazione del tipo:

```c
/* Write chunks of generic data to STREAM. */
extrn size_t fwrite __P ((__const void *__restrict __ptr, 
                       size_t __size, size_t __n, 
                       FILE *__restrict __s));
```

dichiara i ben noti 4 parametri:

---

4In realtà, la collocazione degli header file può variare a seconda della distribuzione utilizzata. Negli esempi che seguono, faremo riferimento alla distribuzione Slackware.
**CAPITOLO 1. PROGRAMMAZIONE DI SISTEMI**

<table>
<thead>
<tr>
<th>Funzione</th>
<th>#include</th>
<th>Descrizione</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen(), fprintf(), ecc.</td>
<td>&lt;stdio.h&gt;</td>
<td>stream I/O</td>
</tr>
<tr>
<td>malloc(), free(), ecc.</td>
<td>&lt;stdlib.h&gt;</td>
<td>gestione memoria, funzioni di controllo</td>
</tr>
<tr>
<td>isalnum(), isdigit(), ecc.</td>
<td>&lt;ctype.h&gt;</td>
<td>elaborazione caratteri</td>
</tr>
<tr>
<td>strcat(), strcmp(), ecc.</td>
<td>&lt;string.h&gt;</td>
<td>elaborazione stringhe, elaborazione aree di memoria</td>
</tr>
<tr>
<td>abs(), rand(), ecc.</td>
<td>&lt;math.h&gt;</td>
<td>funzioni matematiche</td>
</tr>
<tr>
<td>gmtime(), clock(), ecc.</td>
<td>&lt;time.h&gt;</td>
<td>funzioni di temporizzazione</td>
</tr>
<tr>
<td>signal(), raise(), ecc.</td>
<td>&lt;signal.h&gt;</td>
<td>gestione segnali</td>
</tr>
</tbody>
</table>

Tabella 1.1: Funzioni di librerie standard.

1. **__ptr**: un puntatore al primo elemento da scrivere
2. **__size**: la dimensione in caratteri di un singolo elemento
3. **__n**: il numero di elementi da scrivere
4. **__s**: un puntatore allo stream di output

Per comodità, abbiamo indicato nella tabella 1.1 le principali funzioni standardizzate del linguaggio C ed il tipo di dichiarazione `#include` che bisogna premettere nel programma per poterle usare.

**Librerie non standard**

Ogni sistema operativo ed, in particolare Linux, offre diverse librerie non standard e non possiamo pretendere di elencarle tutte. Un semplice esempio sarà però sufficiente ad illustrare le potenzialità d’uso di talune librerie.

Supponiamo di volere realizzare una interfaccia a menu (vedi capitolo 5) tramite un programma C. Le funzioni dello stream I/O non consentono di posizionare il cursore sullo schermo né di leggere caratteri da tastiera, se non dopo avere premuto il tasto di `<INVIO>`. Per ovviare alle suddette limitazioni, sono molto usate le librerie **ncurses** e **SLang**. Le funzioni in esse contenute possono essere invocate facendo uso, rispettivamente delle frasi:

```c
#include <ncurses/ncurses.h>
```
1.3. REALIZZAZIONE DI FUNZIONI DI LIBRERIA: IL RUOLO DELLE API

Ancora una volta, scrivere un programma in ANSI C che faccia uso di librerie non standardizzate ne limita la portabilità: il programma sarà senz’altro compilato correttamente su una piattaforma hardware/software diversa ma non è detto che nel nuovo sistema sia presente la libreria non standard richiesta.

1.3 Realizzazione di funzioni di libreria: il ruolo delle API

Esaminando con attenzione la tabella 1.1, il lettore osserverà che alcune delle funzioni di libreria standard possono tranquillamente essere programmate in C mentre altre non possono esserlo in quanto necessitano di informazioni non disponibili all’interno di tale linguaggio.

Le funzioni aritmetiche, ad esempio, oppure quelle che elaborano caratteri, possono essere realizzate interamente in C. Ma come realizzare in C una funzione come gettimeofday() che restituisce la data e l’ora corrente, oppure una funzione come exit() che causa la terminazione immediata del programma?

La risposta alla suddetta domanda è semplice: numerose funzioni di libreria sono funzioni speciali chiamate API, un acronimo per Application Programming Interface.

Le API non sono altro che funzioni di interfaccia tra il sistema operativo e l’utente che facilitano l’uso da parte del programmatore dei servizi offerti dal sistema operativo. Diamo un paio di esempi per chiarire tale concetto.

Le API malloc() e free() consentono di allocare e de-allocare aree di memoria. Esse effettuano quindi richieste al sistema operativo, rispettivamente, per allocare e de-allocare memoria. Vedremo nel capitolo 6 come ciò viene effettivamente realizzato.

Le API dello stream I/O come fopen(), fprintf(), ecc. effettuano richieste al sistema operativo per avviare od effettuare operazioni di I/O, oltre ad offrire uno schema di bufferizzazione dei dispositivi di I/O coinvolti (stream).
Torniamo ancora una volta, l’ultima, sul discorso della portabilità dei programmi. Le API dipendono dal particolare sistema operativo utilizzato: le API di Windows XP non hanno niente a che vedere con quelle di Linux. Ciò nonostante, la definizione delle funzioni di libreria C per i due suddetti sistemi operativi è la stessa. La portabilità di programmi C è quindi realizzata in pieno, pur lasciando al realizzatore del compilatore C la libertà di implementare le funzioni di libreria facendo uso delle API più adatte tra quelle offerte dal sistema operativo.

1.4 La programmazione di sistemi: una antepri-
ma

Le lezioni ed esercitazioni di questo corso hanno l’intento di illustrare i rudimenti della programmazione di sistema facendo riferimento al linguaggio C ed al sistema operativo Unix (in particolare Linux). Come potete notare dal calendario delle lezioni, le due principali aree considerate saranno la programmazione di I/O e la programmazione concorrente.
Capitolo 2

EVOLUZIONE DEI SISTEMI OPERATIVI

2.1 Introduzione

Lo studio del software di base in generale e dei sistemi operativi in particolare si rivela complesso in quanto doppiamente condizionato sia dalla architettura (hardware) del calcolatore per cui è progettato tale software, sia dalle diverse esigenze dell’utenza a cui il sistema finale (hardware più software) è destinato.

L’approccio seguito in questa dispensa è quello di scindere, per quanto possibile, le esigenze dell’utenza, e quindi le caratteristiche funzionali del software di base, dalle tecniche di realizzazione. A tale scopo, si introducono alcuni concetti preliminari attinenti all’area del software di base. Successivamente, si illustrano, prescindendo dai criteri realizzativi, le funzionalità di varie classi di sistemi operativi mettendo in evidenza, allo stesso tempo, l’evoluzione storica che tali sistemi hanno subito in funzione delle innovazioni tecnologiche conseguite.

2.2 Software di base e sistemi operativi

A differenza di altre macchine, il calcolatore, considerato come congerie di circuiti hardware, non è un oggetto facilmente utilizzabile di per sé stesso: esso è in grado di eseguire istruzioni e quindi programmi, purché tali programmi siano scritti in linguaggio macchina binario, l’unico riconosciuto dal calcolatore. Inoltre, dopo aver premuto il pulsante di accensione, il processore esegue codice registrato in
CAPITOLO 2. EVOLUZIONE DEI SISTEMI OPERATIVI

una memoria non volatile per cercare in una area di memoria (generalmente, su una traccia di disco prefissata) le istruzioni da caricare nella RAM e da eseguire per prime. Come è ovvio, questa inizializzazione della macchina, chiamata bootstraping, sarà effettuata con successo, posto che la traccia di disco letta contenga effettivamente un programma scritto in linguaggio macchina.

Si desume da quanto appena detto che è praticamente impossibile per l’utente utilizzare direttamente l’hardware di un calcolatore, anche accettando di programmare in linguaggio binario, a meno di non disporre di strumenti che gli consentano da un lato, di immettere programmi nella macchina e, dall’altro, di estrarre dalla memoria verso l’esterno i risultati dei calcoli eseguiti.

Fortunatamente, già negli anni ’40 i progettisti delle prime macchine scoprirono che era conveniente sviluppare programmi in maniera modulare introducendo sottoprogrammi (subroutine) di uso comune e richiamandoli dal programma principale: era nato il software di base inteso come corredo di programmi standard accessibili all’utente con lo scopo di facilitare l’interazione uomo/macchina.

Da allora, il software di base è evoluto di pari passo all’hardware con delle marcate interdipendenze: sviluppi nel campo delle architetture e delle apparecchiature periferiche rendono necessario la realizzazione di nuovo software di base; allo stesso tempo, tecniche di programmazione collaudate condizionano la progettazione dei più recenti microprocessori.

Anche se non esistono definizioni rigorose, si indica come software di base tutti i programmi che possono essere utilizzati da utenti diversi e che consentono di sviluppare più agevolmente altri programmi. A differenza dei programmi applicativi che, come il nome indica, servono a specializzare il calcolatore per risolvere problemi legati al mondo della produzione e del lavoro, i programmi nell’area del software di base si collocano a un livello intermedio tra l’hardware e i programmi applicativi. Un programma di gestione magazzino sviluppato per una azienda è un esempio di programma applicativo; viceversa, un programma flessibile che consente, partendo da uno schema generale, di costruire un programma di gestione magazzino, fornendo informazioni specifiche sul modo in cui l’azienda registra i prodotti immagazzinati, può essere considerato software di base.

Si indicano, infine, con il nome di sistemi operativi corredi di programmi messi a punto per calcolatori i quali costituiscono in qualche modo il livello di software di base più vicino all’hardware.

Anche in questo caso, l’area di demarcazione tra sistema operativo e software di base risulta sfumata: un sistema di archiviazione (file system) è considerato come parte essenziale di un sistema operativo; un compilatore può esserlo o meno a seconda dalla presenza o assenza di un supporto di esecuzione (run-time
2.3. CLASSIFICAZIONE DEI SISTEMI OPERATIVI

Risulta opportuno, prima di procedere oltre nello studio dei sistemi operativi, dare una schematica classificazione di tali sistemi che metta in evidenza le caratteristiche dell’utenza che essi devono soddisfare, più che le tecniche realizzative hardware e software. A tale scopo, si distinguono cinque tipi di sistemi operativi chiamati, rispettivamente, dedicati, a lotti, interattivi, transazionali e per il controllo di processi.

I sistemi operativi dei primi tre tipi offrono all’utente la possibilità di scrivere, mettere a punto ed eseguire programmi arbitrari scritti in uno dei linguaggi di programmazione sopportato dal sistema. L’utente tipico di tali sistemi che saranno descritti nel prossimo paragrafo è quindi il programmatore che accede al calcolatore tramite un terminale. Il dialogo utente/sistema si svolge inviando comandi da terminale e ricevendo in risposta messaggi audiovisivi dal sistema. Come si vedrà in seguito, le funzioni del sistema operativo nell’assistere il programmatore vanno ben oltre la semplice compilazione ed esecuzione controllata di programmi per cui si parla comunemente di ambiente di sviluppo offerto dal sistema operativo, intendendo con tale termine l’insieme di servizi offerti che consentono una maggiore facilità nella progettazione, nella stesura e nella verifica di nuovo software.

2.3.1 Sistemi transazionali

I sistemi transazionali sono invece dei sistemi operativi disegnati attorno ad una specifica applicazione: l’utente non è più libero di “inventare” nuovi programmi per poi richiederne l’esecuzione ma può soltanto scegliere da un menù prefissato quale funzione intende eseguire; fatto ciò, egli deve in generale immettere i dati necessari per calcolare la funzione. Esempi tipici di sistemi transazionali sono i sistemi di prenotazione posti o i sistemi per la movimentazione dei conti correnti bancari. Le interazioni dell’utente con il sistema operativo prendono il nome di transazioni (l’annullamento di una prenotazione oppure un versamento in conto corrente sono esempi di transazioni).

Va osservato che tali sistemi non fanno uso di un normale file system bensì di
un Data Base Management System (DBMS) in grado di effettuare operazioni di
Undo nel caso in cui, per qualche motivo, la transazione sia stata solo in parte
eseguita ma non completata.

2.3.2 Sistemi di controllo

Nei casi esaminati in precedenza, si è supposto che il sistema operativo svolga una
funzione di interfaccia tra l’hardware del calcolatore e un operatore al terminale.
Nel caso dei sistemi per il controllo di processi, tale ipotesi viene in parte o del
tutto a cadere: l’obiettivo principale di tali sistemi è quello di acquisire periodi-
camente dati provenienti da sensori e altre apparecchiature remote, di analizzare
tali dati e di emettere quindi, in seguito all’analisi, opportuni segnali ad attuatori
collegati ad apparecchiature di uscita. I sistemi di controllo si dividono a loro
volta in due categorie:

- **sistemi ad anello aperto**: esiste un operatore che analizza i segnali ricevuti
dall’esterno in tempo reale e prende le opportune decisioni.

- **sistemi ad anello chiuso**: non è previsto alcun intervento umano; i segnali
ricevuti dall’esterno sono elaborati e danno luogo a comandi inviati ai vari
dispositivi collegati.

Un sistema di controllo di traffico aereo è ad anello aperto in quanto il controllore
di volo recepisce le immagini inviate dai radar sullo schermo e comunica via radio
a piloti le opportune istruzioni.

I sistemi di controllo per centrali telefoniche elettroniche ed i sistemi di pilotaggio
automatico sono ad anello chiuso in quanto l’operatore umano interviene solo in
casi eccezionali.

I sistemi per il controllo di processi hanno una qualche affinità con i sistemi tran-
sazionali in quanto consentono di svolgere soltanto pochi compiti specifici. Essi,
tuttavia, ricevono i dati di ingresso non da esseri umani ma da sensori collegati
ad apparecchiature di vario tipo. Inoltre, a differenza dei sistemi transazionali,
hanno dei vincoli stringenti per quanto riguarda il tempo massimo disponibile
per acquisire ed elaborare l’informazione proveniente dall’esterno.

I **sistemi embedded** costituiscono un caso particolare di sistemi di controllo ad
anello chiuso. Come il nome indica, oltre a funzionare senza richiedere alcuna
interazione con l’utente, essi sono “annidati” (embedded) in un qualche circuito
elettronico composto da un microprocessore e da appositi sensori ed attuatori,
oltre che da porte di comunicazione per il trasferimento di dati (porta seriale, scheda di rete, ecc.).

I sistemi embedded sono ormai entrati nella nostra vita: i telefoni cellulari, i sistemi ad iniezione elettronica, oppure le automobilini telecomandate sono alcuni esempi tra i tanti tipi di manufatti che fanno uso di sistemi embedded.

Poiché il sistema embedded è un sistema rudimentale che non possiede alcun ambiente di sviluppo, i programmMI per tale sistema si effettuano avvalendosi di un altro calcolatore dotato di un sistema operativo interattivo che dispone di un cross compiler, ossia di un compilatore che genera codice per il processore del sistema embedded. Tale codice viene quindi trasferito nella memoria del sistema collegando il calcolatore con una porta, ad esempio la porta seriale, del sistema embedded. Gli ambienti di sviluppo per sistemi embedded prendono il nome di sistemi di sviluppo.

Concludendo, si ricorda che la classificazione introdotta prescinde dalle caratteristiche hardware del sistema: un home computer con un interprete Basic microprogrammato include un sistema operativo dedicato che offre all’utente un rudimentale ambiente di sviluppo; viceversa, un mainframe con centinaia di terminali collegati può essere utilizzato prevalentemente come sistema transazionale. I collegamenti tra l’architettura del calcolatore e la struttura del sistema operativo saranno invece esaminati in capitoli successivi.

Se il sistema embedded è particolarmente semplice, esso non fa uso di file system e sono sufficienti poche funzioni di ingresso/uscita a posto di un sistema operativo vero e proprio. L’utente realizza in questo caso il proprio sistema operativo.

Se, vice versa, il sistema embedded svolge compiti più complessi, è necessario includere in esso un sistema operativo. I telefoni cellulari in grado di collegarsi in rete necessitano di un sistema operativo dotato di uno stack TCP/IP.

2.4 Evoluzione storica

Per mettere in evidenza la stretta interdipendenza tra l’hardware dei calcolatori e i sistemi operativi, è interessante passare brevemente in rassegna i diversi tipi di colloquio uomo-macchina che sono stati messi a punto dagli albori dell’informatica a oggi, con particolare riferimento ai sistemi dedicati, batch e interattivi.
2.4.1 Sistemi dedicati (prima versione)

La prima fase che si protrae fino all’inizio degli anni ’50 può essere definita come quella dei sistemi dedicati. I vari utenti dispongono a turno dell’intero sistema: durante l’intervallo di tempo concesso (nel lontano 1950, il tempo macchina era una risorsa pregiata concessa a pochi utenti privilegiati per brevi periodi), il programmatore carica programmi in memoria, li fa eseguire e ne controlla l’esecuzione. Per richiamare le poche funzioni del sistema operativo, invia appositi comandi impostando interruttori e commutatori della console (la tastiera non era ancora stata introdotta) mentre, per controllare l’esecuzione del programma, legge i valori di alcuni registri rappresentati sul pannello tramite indicatori luminosi.

Il calcolatore Universal Electronic Computer costruito dall’università di Manchester utilizzava un lettore/perforatore di nastro con caratteri di 5 bit per leggere programmi dall’esterno. Non esisteva linguaggio assemblativo per cui la programmazione era effettuata in una alfabeto a 32 simboli e i caratteri di tale alfabeto erano trasferiti come gruppi di 5 bit in memoria. Come è stato accennato in precedenza, il caricamento di sottoprogrammi si effettuava in modo statico a partire da un indirizzo di base prefissato: si distingueva la “programmazione in grande” che consisteva nel calcolare i vari indirizzi di base dei sottoprogrammi tenendo conto delle loro lunghezze dalla “programmazione in piccolo” che consisteva invece nel mettere a punto un singolo sottoprogramma. Questa seconda attività era generalmente effettuata direttamente durante l’esecuzione del programma: appositi pulsanti e commutatori della console consentivano di fermare l’esecuzione del programma o di rallentarla in modo da consentire al programmatore di leggere i valori assunti dai registri interni della macchina. Il sistema operativo di tale calcolatore consisteva essenzialmente in programmi per pilotare i pochi dispositivi di ingresso/uscita disponibili: il lettore/perforatore di nastro, il monitor rudimentale (veniva visualizzato il contenuto binario di gruppi di parole di memoria), la memoria ausiliaria costituita da un tamburo magnetico.

2.4.2 Gestione a lotti

La gestione appena descritta non era soddisfacente; il calcolatore considerato come macchina costosa e sempre più potente doveva essere sfruttato meglio: in particolare i periodi in cui l’utente bloccava la macchina per meglio controllare l’esecuzione del suo programma portavano ad una scarsa percentuale di utilizzazione della CPU. La risposta a tale problema venne fornita da un lato dalla introduzione di lettori di schede perforate e dall’altro da un primo affinamento del sistema operativo: nacque così all’inizio degli anni ’50 una modalità di colloquio
2.4. **EVOLUZIONE STORICA**

chiamata gestione a lotti (batch processing) che inibiva in pratica all’utente l’uso della console. Le principali caratteristiche di tale forma di colloquio possono così essere sintetizzate.

- Le richieste degli utenti sono costituite da pacchi di schede perforate; ogni pacco, chiamato lavoro o job inizia con una scheda di identificazione dell’utente e consiste in una sequenza di passi o step.

Ogni step rappresenta una richiesta svolta dall’utente al sistema operativo per ottenere un servizio elementare generalmente associato alla esecuzione di un programma. Esempi di step sono la compilazione di un programma, il linkaggio di un gruppo di programmi compilati in precedenza, oppure l’esecuzione di un programma già compilato e linkato in precedenza. La scheda di identificazione del lavoro e le altre schede che servono a definire i vari step sono chiamate schede di controllo. Anche i programmi e gli eventuali dati devono apparire nel lavoro che consiste quindi in una alternanza di schede di controllo e di schede dati; queste ultime possono contenere un programma da compilare oppure delle informazioni alfanumeriche che saranno usate durante l’esecuzione del programma.

- Eseguito un lavoro, l’utente ottiene dal sistema operativo un tabulato dove sono stampati consecutivamente i risultati prodotti dai vari step eseguiti. Poiché non è più possibile accedere alla console, il sistema operativo deve registrare in tale tabulato, oltre agli eventuali risultati, tutti i messaggi e diagnostici che si sono verificati durante l’esecuzione del lavoro.

- Il sistema operativo include un programma di controllo chiamato batch monitor che svolge le seguenti funzioni:
  - Legge da lettore di schede il prossimo lavoro. Se il lettore di schede è vuoto, rimane in attesa di un segnale dell’operatore che indica che sono state collocate altre schede sul lettore.
  - Esegue successivamente gli step del lavoro letto e provvede, ad ogni step, a caricare da nastro magnetico o da disco l’apposito programma per poi passare ad esso il controllo.
  - Se si verifica un errore durante l’esecuzione di uno step, il batch monitor stampa un messaggio di diagnostica, salta i rimanenti step del job e passa a considerare il job successivo.
Usando lo schema appena descritto, ogni step di un job consiste nel leggere schede, eseguire istruzioni e stampare risultati. Poiché il lettore di schede e la stampante sono lente rispetto alla CPU, si verifica frequentemente durante l’esecuzione di uno step che la CPU rimanga bloccata in attesa di segnali del tipo “fine lettura schede” o “fine scrittura riga di stampa”. Per rimuovere tale collo di bottiglia furono quindi introdotti perfezionamenti nella gestione a lotti. Tali perfezionamenti furono resi possibili dalla introduzione di nuove unità periferiche e di calcolatori dotati di più processori autonomi.

In una prima fase, con l’introduzione di lettori di nastro magnetico aventi un elevato tasso di trasferimento, furono introdotte architetture che prevedevano sistemi ausiliari per effettuare la conversione scheda-nastro e quella nastro-scheda. Il calcolatore principale non interagiva più con periferiche lente ma solo con unità a nastro mentre i calcolatori ausiliari svolgevano in parallelo le attività di conversione.

Anche se serviva a rimuovere il collo di bottiglia citato in precedenza, l’uso di elaboratori ausiliari poneva problemi di gestione: il numero di nastri da montare e smontare era notevole e gli errori degli operatori frequenti.

Il perfezionamento successivo si ebbe verso la fine degli anni ‘50 quando apparvero i primi calcolatori con canali di I/O autonomi nonché le prime memorie a disco di grande capacità: in pratica, venne ripresa l’idea precedente di fare interagire la CPU prevalentemente con unità periferiche veloci, in questo caso i dischi. Tuttavia, grazie ai canali di I/O integrati nel calcolatore, non era più necessario ricorrere a calcolatori ausiliari in quanto tali canali sono dei processori specializzati in grado leggere schede, stampare dati o interagire coi dischi, interferendo in modo ridotto con le attività della CPU (esistono in realtà interferenze dovute agli accessi simultanei alla memoria da parte dei vari processori e al fatto che la sincronizzazione canale di I/O-CPU avviene tramite interruzioni che richiedono, a loro volta, l’esecuzione sulla CPU di brevi programmi di gestione; comunque, tali interferenze possono essere alquanto contenute in un sistema ben dimensionato). La soluzione tuttora adottata ha portato all’uso di dischi di spooling (Simultaneous Peripheral Operations On Line): il disco di spooling di ingresso contiene “schede virtuali”, ossia schede lette dal lettore e contenenti job non ancora eseguiti, quello di spooling di uscita contiene invece “tabulati virtuali”, ossia tabulati già prodotti relativi a job eseguiti ma non ancora stampati. I canali di I/O provvedono a riempire o svuotare i dischi di spooling e a gestire i trasferimenti tra memoria e dischi mentre la CPU può essere quasi interamente dedicata ad eseguire programmi degli utenti o del sistema operativo.

Verso la fine degli anni ‘60 la gestione a lotti subisce, grazie ai progressi conseguiti nel campo dell’hardware e delle telecomunicazioni, una importante e radicale evo-
luzione consistente nell’eliminare l’ormai antiquato lettore di schede, consentendo agli utenti di sottomettere job da terminali remoti.

L’introduzione di circuiti integrati e la conseguente diminuzione dei costi dei componenti hardware fa sì che è ormai possibile servirsi di “terminali intelligenti”, sia per editare un job da inviare successivamente ad un mainframe perché venga eseguito con le modalità della gestione a lotti, sia per visualizzare, archiviare o stampare i risultati del job eseguito. Lo sviluppo di reti di trasmissione dati ha inoltre consentito di distribuire i punti di accesso al grande calcolatore, ampliando in questo modo l’utenza potenziale.

Attualmente, emergono due aree di applicazioni in cui la gestione a lotti si rivela più conveniente di quella interattiva (vedi sottoparagrafo successivo): in primo luogo, negli ambienti di produzione in cui sono utilizzate in modo periodico, e quindi prevedibile, procedure complesse che impegnano il calcolatore per delle ore consecutive, la gestione a lotti, oltre a consentire di sfruttare appieno la potenza di calcolo del sistema, consente anche di stabilire un calendario di esecuzione (schedule) dei job e dà quindi al gestore del centro di calcolo la certezza che un determinato job sarà eseguito entro una data prefissata.

In secondo luogo, la gestione a lotti rimane la più adatta ogni qualvolta la risorsa di calcolo ha un costo molto elevato: questo si verifica per i supercalcolatori utilizzati per eseguire calcoli particolarmente impegnativi. In tale caso, il costo di simili macchine suggerisce di ottimizzarne l’uso facendo in modo che i programmi degli utenti siano già messi a punto su macchine meno costose e ricorrendo alla gestione a lotti per sfruttare appieno la potenza del supercalcolatore.

2.4.3 Sistemi interattivi

Per quanto efficiente, la gestione a lotti non è la più adatta in ambienti in cui lo sviluppo e la messa a punto di programmi sono considerate attività prioritarie: in questo caso, infatti, il dovere attendere che termini il job precedente prima di potere effettuare una nuova prova limita notevolmente la produttività dei programmatori. Per ovviare a tale inconveniente, fu sviluppato nel 1963 presso il Massachusetts Institute of Technology (MIT) il sistema operativo Compatible Time Sharing System (CTSS) basato su un elaboratore IBM 7090 modificato con ulteriori 32 KB di RAM ed un circuito timer in grado di emettere segnali di interruzione ad una frequenza costante. Il CTSS può essere considerato il primo sistema operativo multiutente di tipo interattivo.

L’idea di base dei sistemi interattivi, anche chiamati a partizione di tempo (in inglese, time-sharing), è quella di assicurare, ad ogni utente collegato tramite
CAPITOLO 2. EVOLUZIONE DEI SISTEMI OPERATIVI

terminale, una frazione garantita delle risorse del calcolatore, a prescindere dalle richieste degli altri utenti collegati. Questo si ottiene tramite un multiplamento nel tempo della CPU tra i vari programmi associati ai terminali attivi: ogni programma ha diritto a impegnare la CPU per un quanto di tempo (tipicamente, si usano quanti di poche centinaia di millisecondi, anche se sarebbe più corretto misurare il quanto in termini di numero di istruzioni eseguibili durante un quanto). Se il programma non termina entro il quanto prefissato, esso viene posto in lista d’attesa ed un nuovo programma riceve il quanto successivo. Se vi sono N terminali attivi, ogni utente ha la certezza di ottenere un quanto di tempo ogni N, qualunque siano i tempi di esecuzione degli altri programmi.

A prima vista, si potrebbe pensare che un sistema interattivo appaia agli N utenti collegati come un calcolatore N volte più lento di quello reale; in pratica, le prestazioni sono molto migliori poiché, durante l’attività di messa a punto di programmi, l’utente alterna fasi di editing del programma nonché fasi di analisi dei risultati e ripensamenti (think time) a fasi di compilazione e esecuzione per cui una buona parte delle richieste degli utenti (editing, presentazione dei risultati) può essere esaudita prevalentemente dai canali di I/O con scarso impegno della CPU.

I sistemi interattivi hanno avuto un notevole impatto nello sviluppo dei sistemi operativi: è stato necessario predisporre schemi hardware e software per proteggere i programmi e gli archivi degli utenti attivi nel sistema da danni causati da programmi non affidabili (o indiscreti?) di altri utenti. Allo stesso tempo, la gestione rigida della CPU basata sull’assegnazione di quanti di tempo di lunghezza prefissata è evoluta per tenere conto del fatto che il sistema operativo non è in grado di conoscere a priori quanto tempo di CPU richiederà un programma. Per questo motivo, si preferisce usare una strategia flessibile: quando un programma inizia ad eseguire, esso è considerato di tipo interattivo, se dopo alcuni quanti di tempo, esso non ha ancora terminato, la sua priorità viene diminuita; infine, se oltrepassa un numero prefissato di quanti, esso viene declassato a lavoro di tipo differito e, in pratica, sarà eseguito come in un sistema di gestione a lotti assieme ad altri lavori dello stesso tipo quando non vi sono più richieste di utenti ai terminali.

La flessibilità e la sicurezza offerta dai moderni sistemi interattivi fa sì che i server specializzati oggi in uso, ad esempio i Web server o gli SQL server, siano basati su tale tipo di sistema operativo. Tipicamente, sono collegati due utenti diversi ad un server: quello specializzato che riceve le richieste specifiche, interroga la base di dati ed invia le risposte, e l’amministratore di sistema che è in grado di intervenire tempestivamente in caso di malfunzionamenti.

Vice versa, l’avvento dei personal computer a basso costo ha reso meno interes-
sante per l’utente comune l’accesso ad un sistema interattivo, anche se, la maggior parte dei sistemi operativi per PC consentono la multiutenza.

2.4.4 Sistemi dedicati (seconda versione)

All’inizio degli anni ’80, appaiono i primi personal computer e con essi si verifica un rifiorire dei sistemi operativi dedicati che erano scomparsi dallo scenario informatico subito dopo l’avvento dei primi sistemi di gestione a lotti. In effetti, l’evoluzione tecnologica consente di produrre oggi a costi contenuti dei calcolatori che, pure avendo un ingombro limitato, hanno una potenza di calcolo paragonabile a quella dei mainframe degli anni ’70. In simili condizioni, ha di nuovo senso impostare un sistema operativo dedicato che serve un unico posto di lavoro poiché l’efficiente utilizzazione del calcolatore cessa di essere un obiettivo prioritario. Paradossalmente, questa nuova generazione di sistemi operativi offre all’utente una interfaccia che risulta per alcuni aspetti superiore a quella offerta da sistemi interattivi sviluppati per mainframe. In particolare, la disponibilità di uno schermo grafico ad alta risoluzione accoppiato all’uso di microprocessori sempre più potenti ha reso possibile la realizzazione di software grafico interattivo che offre affascinanti possibilità di utilizzazione. Il sistema operativo MacOS della Apple è uno dei capostipiti più noti di questa nuova generazione di sistemi operativi dedicati.

2.5 Virtualizzatori

Accenniamo in questo breve paragrafo ad un tipo di software di base che, pur non essendo un sistema operativo vero e proprio, interagisce coi sistemi operativi.

Si tratta di virtualizzatori di diverso tipo che consentono di gestire in partizione di tempo più sistemi operativi di tipo diverso i quali condividono le stesse risorse hardware. L’idea di base consiste nell’intercettare gli accessi ai dispositivi di I/O eseguiti dai sistemi operativi ospite effettuandoli tramite il codice del virtualizzatore.

Il primo virtualizzatore chiamato CP/CMS è stato realizzato dalla IBM nel 1966. Oggi sono disponibili diversi virtualizzatori per personal computer: VMware, OpenVX, Xen, ecc.

La realizzazione di virtualizzatori risulta più agevole quando l’hardware del calcolatore dispone di almeno tre livelli di protezione: il livello 0 è riservato al
CAPITOLO 2. EVOLUZIONE DEI SISTEMI OPERATIVI

virtualizzatore, i kernel ed i processi di sistema dei vari sistemi operativi eseguo-
no con livello di protezione 1 ed i programmi applicativi eseguono con livello di
protezione 2.

2.6 Sviluppi futuri

È sempre difficile fare previsioni attendibili in settori in rapida evoluzione come
quello informatico. Tuttavia, molti pensano che nel prossimo decennio si con-
soliderà l’affermazione dei personal computer e che, in conseguenza, i maggiori
sforzi dei progettisti saranno rivolti alla progettazione di sistemi operativi per
tale classe di macchine.

Probabilmente, i risultati più interessanti si conseguiranno nel settore dei sistemi
dedicati in rete. Con tale termine, si intende un sistema dedicato che consente non
soltanto di utilizzare il personal localmente ma anche di mantenerlo collegato in
rete (ad esempio, una rete pubblica di trasmissione dati quale Internet). In questo
modo, l’utente può operare localmente mentre, simultaneamente e con un minimo
di interferenza, il sistema operativo provvede a inviare messaggi già preparati
in precedenza a nodi remoti e, allo stesso tempo, riceve e memorizza su disco
messaggi inviati da altri nodi della rete al personal. Un altro promettente settore
riguarda il collegamento tramite linea dedicata veloce di un elevato numero di
personal computer. Disponendo di una linea veloce di qualche Gigabyte/sec
e delle opportune schede di interfaccia mediante le quali collegare i personal
computer alla linea, l’intero sistema risulta assimilabile ad un calcolatore parallelo
con un rapporto prestazioni/costo molto interessante, tenuto conto delle economie
di scala realizzate operando nel settore dei personal computer.
Capitolo 3

SOFTWARE DI BASE

3.1 I componenti di un sistema operativo

Tecnicamente parlando, un sistema operativo consiste essenzialmente in quattro parti principali:

- un programma di inizializzazione che viene eseguito quando viene avviato l'elaboratore;
- un NUCLEO che esegue chiamate di sistema, ossia particolari funzioni mediante le quali i programmi possono richiedere uno dei vari servizi offerti dal NUCLEO (tratteremo questo argomento nel paragrafo 6.4);
- un sistema d’archiviazione (file system);
- un’interfaccia che consente all’utente di esprimere le proprie richieste.

In effetti, i testi di sistemi operativi più accreditati si limitano a trattare i suddetti argomenti. Dato il carattere introduttivo di queste dispense, riteniamo tuttavia opportuno studiare i sistemi operativi seguendo un approccio “top down”, per cui iniziamo col descrivere le caratteristiche principali di alcuni importanti programmi di utilità (viene anche usato il termine software di base). Come vedremo nelle prossime pagine, tali programmi risultano indispensabili ai programmatori, agli utenti generici e agli amministratori di sistema.
3.2 Messa a punto di programmi

Iniziamo col parlare degli strumenti che storicamente sono stati quelli introdotti per primi ed il cui obiettivo è quello di facilitare l’utente nel realizzare programmi.

3.2.1 Assemblatori, compilatori e interpreti

E’ stato ribadito, a proposito del software di base, che l’unico linguaggio riconosciuto dal calcolatore è il linguaggio macchina binario. Poiché è estremamente disagevole scrivere programmi servendosi di tale linguaggio, sono stati sviluppati programmi traduttori il cui compito è quello di trasformare programmi scritti in uno dei vari linguaggi di programmazione esistenti (il cosiddetto *linguaggio sorgente*) in programmi scritti in linguaggio macchina.

Il processo di traduzione ha caratteristiche diverse, a seconda del linguaggio sorgente e delle modalità di realizzazione. In generale, la traduzione precede l’esecuzione del programma. In questo caso, se le frasi eseguibili del linguaggio sorgente corrispondono alle istruzioni del calcolatore, il traduttore prende il nome di *assemblatore.* Nel caso contrario, i programmi traduttori sono chiamati *compilatori.*

Il linguaggio assemblativo per i microprocessori Intel 80x86, ad esempio, è un linguaggio sorgente simile al linguaggio macchina di tali microprocessori; esso prevede frasi eseguibili del tipo *add* oppure *mov* corrispondenti al repertorio di istruzioni riconosciuto da tali calcolatori. Viceversa, il linguaggio C è un linguaggio sorgente più potente per cui ogni frase eseguibile deve essere tradotta dal compilatore in sequenze di istruzioni in linguaggio macchina (si pensi, ad esempio, alle frasi C che includono espressioni aritmetiche di arbitraria complessità).

In altri casi, la traduzione può andare di pari passo con l’esecuzione del programma: traduttori di tale tipo sono chiamati *interpreti.* L’interprete traduce una frase alla volta del programma sorgente e la esegue immediatamente prima di passare a tradurre la frase successiva.

Un programma interpretato ha tempi di esecuzione molto maggiori di uno compilato, proprio perché ogni frase del linguaggio sorgente ed ogni variabile deve essere ripetutamente tradotta e riesaminata dall’interprete quando inclusa in un ciclo di programma. In prima approssimazione, un programma interpretato ha un tempo di esecuzione circa 50 volte maggiore di quello di un programma compilato. In compenso, gli interpreti occupano meno memoria dei compilatori e offrono una ottima diagnostica durante l’esecuzione per cui sono molto usati in
piccoli sistemi e nello sviluppo di linguaggi specializzati. Tra gli interpreti più noti citiamo quello per il linguaggio Basic e tutti gli interpreti di comandi (vedi sezione 5.3).

Dal punto di vista del sistema operativo, gli assemblatori e i compilatori sono dei programmi eseguibili scritti in linguaggio macchina che richiedono per la loro esecuzione un file di ingresso, uno di uscita e alcuni file ausiliari temporanei, ossia eliminabili dopo che è terminata l’esecuzione del programma compilatore. Il file d’ingresso contiene il programma sorgente preparato in precedenza servendosi di un opportuno editor. Il file di uscita contiene invece un modulo oggetto (sono anche usati i termini: programma rilocabile e programma oggetto). Tale programma è scritto in linguaggio macchina ma non è ancora eseguibile in quanto, solitamente, fa riferimento ad altri programmi e dati esterni, ossia non inclusi nel programma sorgente, che il compilatore non è in grado di tradurre.

Alcuni compilatori più sofisticati prevedono più fasi consecutive. Il “GNU C compiler”, ad esempio, prevede tre fasi distinte:

1. **preprocessamento**: frasi del tipo:
   ```
   #include <nomefile>
   ```
   oppure:
   ```
   #ifdef <variabile_compilazione>.... #endif
   ```
   sono tradotte nelle opportune frasi C;

2. **compilazione**: il programma C viene tradotto nel linguaggio assemblativo del processore utilizzato;

3. **assemblaggio**: il programma assemblativo viene tradotto in linguaggio macchina dando luogo ad un modulo oggetto.

Questo approccio consente al programmatore di inserire sequenze di istruzioni in linguaggio assemblativo all’interno del programma sorgente. Il GNU C compiler, ad esempio, riconosce una macro chiamata `asm()` che consente di inserire frasi in linguaggio assemblativo all’interno del programma C. Questa tecnica è chiamata *assembly inline*.

Un tipico caso dove l’uso del linguaggio assemblativo risulta indispensabile è il seguente: tutti i programmi che pilotano apparecchiature periferiche devono potere accedere ad opportune porte di I/O accessibili nell’architettura 80x86 mediante apposite istruzioni di tipo `in` e `out` (vedi capitolo 11). Il sistema operativo offre
CAPITOLO 3. SOFTWARE DI BASE

quindi opportune funzioni C per accedere a tali porte ed ognuna di queste funzioni incorpora istruzioni assembler poiché il linguaggio C ignora l’esistenza di porte di I/O.

Per fortuna, solo una limitata parte del sistema operativo deve necessariamente essere codificata in linguaggio assemblativo. La rimanente parte, oltre il 90% del codice, può essere scritta in un linguaggio ad alto livello quale il C.

3.3 Linkaggio di programmi

Il linkaggio di programmi si rivela indispensabile in applicazioni complesse dove è facile superare le decine di migliaia di frasi: in tali casi, non è conveniente registrare l’intero programma sorgente in un unico file ma è di gran lunga preferibile decomporre l’applicazione e registrare i programmi sorgente che la compongono in vari file sorgente ognuno dei quali può essere compilato separatamente.

La nozione di file sorgente dipende dal linguaggio di programmazione: in un linguaggio quale il C, qualunque funzione o gruppo di funzioni può essere registrata in un file sorgente. In modo analogo, è possibile registrare in un file sorgente gruppi di variabili globali o qualunque combinazione di funzioni e variabili globali. Come è ovvio, deve sempre essere presente un file sorgente che contiene la funzione main().


Nei sistemi operativi di tipo Unix, i moduli oggetto sono contrassegnati dall’estensione .o: prog.c denota, ad esempio, un file sorgente scritto in C mentre prog.o denota il modulo oggetto ottenuto compilando prog.c tramite il comando:

```
gcc -c prog.c
```

In assenza di ulteriori informazioni, il compilatore non sa quale indirizzo iniziale assegnare al modulo oggetto per cui sceglie un indirizzo arbitrario, solitamente 0. Il modulo dovrà successivamente essere “rilocato” assegnando un indirizzo iniziale diverso da 0. Ciò viene fatto in fase di linkaggio.
Nel produrre un modulo oggetto, il compilatore evita quindi di fare uso di indirizzi assoluti. Esso genera invece codice rilocabile facendo uso delle seguenti tecniche:

- tutti gli indirizzi inclusi in istruzioni di salto all’interno della stessa funzione sono espressi come offset (spiazzamenti) rispetto all’indirizzo dell’istruzione eseguita;
- gli indirizzi di variabili locali appartenenti ad una delle funzioni C non sono mai assoluti in quanto il compilatore alloca tali variabili in cima allo stack, per cui si indirizzano tramite l’offset rispetto al registro puntatore allo stack (registro esp nell’architettura Intel 80x86);
- gli indirizzi di variabili globali definite nel file sorgente o in altri file sorgente non sono risolti dal compilatore (vedi dopo);
- gli indirizzi di funzioni, ossia i parametri di istruzioni `call`, definite nel file sorgente o in altri file sorgente non sono risolti dal compilatore (vedi dopo).

I moduli oggetto non sono logicamente indipendenti tra loro: il modulo A può includere una chiamata ad una funzione contenuta in un altro modulo B, ignoto al momento della compilazione di A. Allo stesso modo, i moduli interagiscono su variabili o strutture di dati comuni contenute in moduli diversi da quello attualmente compilato. Una conseguenza della decomposizione in moduli è quindi che il compilatore non è in grado di associare un indirizzo rilocabile ad ogni funzione e ad ogni variabile contenuta in un modulo: esso esegue una traduzione parziale, creando quindi una tabella dei simboli esterni per tutti quei nomi che non ha trovato all’interno del modulo.

Nel linguaggio C, la dichiarazione `extern` deve essere usata dal programmatore per contrassegnare tutte le variabili e funzioni che non sono definite all’interno del file sorgente. In generale, una funzione o variabile di tipo `extern` può essere definita:

1. in un altro file sorgente preparato dal programmatore;
2. in una libreria di programmi messa a punto dal programmatore;
3. in una libreria di programmi (standard o non standard) inclusa in una delle directory del sistema operativo.

Le funzioni e variabili esterne possono essere linkate in modo statico oppure dinamico, a seconda delle opzioni di linkaggio scelte dall’utente (vedi dopo).

Diamo ora una descrizione molto sommaria di come sono effettuati i due tipi di linkaggio.
3.3.1 Linkaggio statico

La figura 3.1 illustra un semplice esempio di linkaggio statico tra due moduli: il modulo A include una chiamata alla funzione `funz()` inclusa nel modulo B. Il modulo B, a sua volta, fa uso di una variabile globale N inclusa nel modulo A.

Quando viene compilato il modulo A, il compilatore non è in grado di “risolvere” l’etichetta `funz`, anche se la dichiarazione del tipo `extern int funz(...)` presente nel codice lo aiuta nello stabilire che `funz` è effettivamente un nome esterno.

Il compilatore inserisce quindi nella tabella dei simboli, ossia in una apposita struttura di dati inclusa in testa al modulo oggetto, il simbolo esterno `funz` nonché gli indirizzi in cui tale etichetta appare. Di solito, infatti, la stessa funzione può essere invocata più volte in parti diverse del programma, per cui è necessario identificare tutti gli indirizzi in cui la funzione viene invocata in modo da poterli sostituire con l’indirizzo valido in una fase della messa a punto del programma successiva alla compilazione.

In modo analogo, la stessa variabile globale N di tipo `extern` può essere usata in parti diverse del programma ed è necessario specificare nella tabella dei simboli in quali istruzioni del programma essa viene usata.

Il compilatore inserisce inoltre nella tabella dei simboli i simboli interni, ossia i nomi di tutte le funzioni e variabili globali incluse nel modulo di compilazione. A differenza dei simboli esterni, tali simboli potranno essere usati dal linker per risolvere indirizzi inclusi in altri moduli oggetto. Nel nostro esempio, il simbolo interno N servirà a risolvere il simbolo esterno N presente nella tabella dei simboli del modulo B.

Lo stesso avviene durante la compilazione del modulo B. La tabella dei simboli di tale modulo oggetto include come simbolo interno l’indirizzo della funzione `funz` ed include come simbolo esterno il simbolo `extern`. Per semplicità si è supposto nella figura che i simboli esterni siano usati soltanto una volta.

Una volta compilati i vari moduli, è quindi necessario fare uso di un nuovo componente del sistema operativo chiamato linker per “cucire” tra loro i vari moduli. Servendosi delle indicazioni contenute nelle tabelle dei simboli incluse in testa ad ogni modulo oggetto, il linker fonde i vari moduli oggetto in un unico file chiamato file eseguibile e completa all’interno di tale file la compilazione inserendo gli indirizzi mancanti nelle varie parti del codice.

Il file eseguibile appena costruito è pronto ad essere caricato in memoria a par-
3.3. LINKAGGIO DI PROGRAMMI

Figura 3.1: Un esempio di riferimenti tra moduli.

Figura 3.2: Un semplice esempio di linkaggio statico tra due moduli.
tire dall’indirizzo impostato dal linker. Basandoci sull’esempio precedente, possiamo vedere nella figura 3.2 come risulta il file eseguibile prodotto dal linker in corrispondenza ai due moduli A e B. I due simboli esterni funz ed N sono stati risolti dal linker: nel primo caso si è usato l’offset 100 rispetto al valore attuale del contatore programma eip. Nel secondo caso, si è usato l’indirizzo assoluto 0x10000400. Il programma risultante non è quindi rilocabile ma deve essere caricato in memoria a partire da un indirizzo prefissato, ossia dall’indirizzo 0x10000000.

Questo approccio può risultare sufficiente per sistemi operativi alquanto semplici, ad esempio per preparare un file eseguibile da caricare in un embedded system. Nei sistemi operativi di tipo multitasking invece, il linker non assegna al file eseguibile indirizzi fisici bensì indirizzi lineari o indirizzi logici. Durante l’esecuzione di un programma, appositi circuiti hardware della CPU traducono gli indirizzi lineari presenti nelle istruzioni da eseguire in indirizzi fisici (vedi capitolo 8).

### 3.3.2 Linkaggio dinamico

Nel caso del linkaggio statico la tabella dei simboli non risolti del file eseguibile risulta vuota in quanto tutti i simboli esterni sono stati risolti nella fase di linkaggio, prima del caricamento del programma.

Solitamente si preferisce però fare uso del linkaggio dinamico che consiste nel delegare ad un componente del sistema operativo chiamato linker dinamico il compito di effettuare il linkaggio ogni volta che il programma in esecuzione chiama una funzione (o accede ad una variabile) il cui indirizzo non è stato risolto in precedenza. In effetti, a meno che il programmatore non specifichi l’opzione -static, esso viene sempre effettuato in modo dinamico.

Illustriamo a grandi linee come avviene questo secondo tipo di linkaggio che si basa su una stretta interazione tra il programma dell’utente e il linker dinamico. Supponiamo per semplicità che il programma eseguibile includa una chiamata ad una singola funzione funz il cui indirizzo non è stato risolto dal linker statico in quanto funz è inclusa in una libreria dinamica (vedi paragrafo 3.3.3) e pertanto deve essere linkata dinamicamente quando il programma la invocherà per la prima volta.

Il compito del linker dinamico è quello di trovare in una delle librerie specificate dall’utente la funzione esterna richiesta dal programma. Trovato l’indirizzo di tale funzione, il linker dinamico non scrive l’indirizzo nell’istruzione call che la invoca poiché, così facendo, modificherebbe il codice eseguibile, cosa da evitare per motivi di sicurezza.
La figura 3.3 illustra come si svolge il linkaggio dinamico. Il programmatore ha provveduto a trasmettere nella riga di comando i nomi di tutte le librerie dinamiche utilizzate dal suo programma. Il linker statico ha preparato una tabella dei simboli contenente i nomi delle funzioni esterne e gli offset nel programma in cui tali nomi sono usati. Esso ha inoltre creato, in aggiunta alla sezione codice ed alla sezione dati del programma, una ulteriore sezione dati chiamata *Global Offset Table (GOT)* ed una ulteriore sezione codice chiamata *Procedure Linkage Table (PLT)*.

Il linker statico ha risolto la chiamata a *funz()* tramite una chiamata con indirezione (*call * *) alla voce della PLT corrispondente a *funz()*.

La PLT include per ogni funzione esterna *funz* un gruppo di tre istruzioni:

1. una *jmp* con indirizzamento indiretto per saltare alla voce della GOT associata al simbolo da risolvere; tale voce contiene inizialmente l’indirizzo della seconda istruzione della voce corrispondente della PLT, ossia l’indirizzo della *push*;

2. una *push* per salvare nello stack l’indice nella tabella dei simboli della voce contenente il simbolo da risolvere;

3. una *jmp* per saltare al linker dinamico *dlresolve* di Linux.
CAPITOLO 3. SOFTWARE DI BASE

Nella prima chiamata, viene eseguita la prima jmp della PLT, quindi la push ed infine viene eseguito dl resolve. Tale programma ricava prima l’indirizzo della funzione e lo scrive nella voce della GOT corrispondente a funz (ciò è consentito perché la GOT è collocata in una sezione dati); successivamente passa il controllo alla prima istruzione di funz() (vedi figura 3.3).

Grazie a tale schema, ogni successiva invocazione alla stessa funzione sarà realizzata tramite una prima jmp alla PLT seguita da una seconda jmp con indirizzamento indiretto nella voce della GOT che contiene ormai l’indirizzo di funz() senza dover più invocare il linker dinamico.

3.3.3 Librerie statiche e dinamiche

Come abbiamo già visto nel paragrafo 1.2, l’esistenza di un linker offre ai programmatori la possibilità di ricorrere a funzioni già messe a punto da altri, oppure di raggruppare alcune funzioni del proprio programma in apposite librerie. Tutti linguaggi di programmazione fanno un ampio uso di tali librerie, per cui non è addirittura possibile eseguire alcuni programmi in un qualsiasi linguaggio se non si effettua il linkaggio con numerose altre funzioni incluse in una o più librerie.

Come è ovvio, ogni linguaggio di programmazione ha le sue apposite librerie, per cui vi saranno librerie di funzioni (o classi nel caso di linguaggi orientati ad oggetti) per il linguaggio C, per il C++, per il Java e così via.

Nell’effettuare il linkaggio tra le funzioni C scritte dall’utente e quelle già presenti in apposite librerie, il linker può fare uso di due tipi di librerie, ognuna delle quali presenta vantaggi e svantaggi.

- **libreria statica**: contiene il codice oggetto di funzioni; il linker costruisce un file eseguibile contenente il codice oggetto corrispondente alla compilazione dei programmi dell’utente nonché il codice di tutte le funzioni di librerie utilizzate. Il programma risultante è autosufficiente, nel senso che tutti i moduli oggetto indirizzati sono contenuti nello stesso file eseguibile. Lo svantaggio di tale approccio è che la stessa funzione di libreria viene duplicata in vari file eseguibili, per cui lo spazio su disco viene sfruttato in modo poco efficace. Il compito del loader risulta alquanto semplice, proprio perché il file eseguibile è autosufficiente.

- **libreria dinamica**: il linker si limita ad inserire nella testata del file eseguibile corrispondente alla compilazione dei programmi dell’utente i nomi delle funzioni di libreria utilizzate e gli indirizzi nel codice in cui compaiono chiamate a tali funzioni. Durante l’esecuzione del programma, le funzioni
incluse nelle librerie sono caricate dinamicamente in una apposita area di memoria (a meno che non siano già state caricate in precedenza) e viene effettuato un linking dinamico tra le istruzioni del programma utente e gli indirizzi delle funzioni richieste.

Il vero problema è tuttavia un altro: le librerie di programma evolvono nel tempo ed i parametri usati da alcune funzioni della libreria possono cambiare da una versione all'altra. Quando ciò avviene, può succedere in alcuni sistemi operativi che l’applicazione selezionata dall’utente non possa essere eseguita su un determinato sistema poiché esso fa uso di una versione della libreria incompatibile con quella richiesta dall’applicazione.

Nel file system di Unix la cartella /bin contiene file di uso comune che possono essere eseguiti da utenti comuni mentre la cartella /sbin contiene file eseguibili che possono eseguiti da utenti superuser.

Le cartelle /usr/local/bin e /usr/local/sbin dovrebbero essere usate dai sistemisti per inserirvi ulteriori file eseguibili dei due tipi. Le librerie di programmi di uso comune sono di solito inserite nella cartella /lib.

Ulteriori librerie specializzate sono inserite nella cartella /usr, anche se non esiste ancora uno standard che specifica, per ogni tipo di libreria, la corretta collocazione nell’albero delle cartelle. Un semplice esempio può servire ad illustrare l’affermazione precedente.

Nelle distribuzioni di Linux, la libreria di funzioni per l’interfaccia grafica X Window ha il nome di percorso /usr/X11R6/lib. Altre librerie specializzate, quali quella che contiene le funzioni attinenti alla rappresentazione di immagini nel formato jpeg, sono incluse nella cartella /usr/local/lib. In altri casi ancora, diversi programmi applicativi codificati tramite numerosi moduli fanno uso di librerie proprie aggiuntive che devono essere inserite in apposite cartelle. Il browser firefox, ad esempio, richiede alcune libreria di funzioni che devono essere inserite nella cartella /usr/lib/firefox-w.x.y.z.

In generale, quella che può sembrare una eccessiva rigidità dell’applicazione, ossia il dovere inserire una libreria di funzioni con un nome di percorso prefissato, si traduce in un notevole beneficio per l’utente. Grazie a tale approccio, è possibile inserire nei programmi applicativi linkati in modo dinamico appositi test per verificare se le librerie dinamiche installate nel sistema sono compatibili con l’applicazione da installare, segnalando eventuali incompatibilità all’utente.

Poiché ogni libreria ha un apposito nome di percorso, risulta inoltre possibile, utilizzare diverse versioni della stessa libreria: questo può risultare utile per eseguire
sia applicazioni più antiche che fanno uso di versioni precedenti di una libreria che applicazioni più recenti che invece richiedono una versione più recente ed incompatibile con quella precedente.

### 3.3.4 Immissione di programmi

Un programma è un documento immesso dall’utente tramite tastiera che consiste in una stringa di caratteri. La funzione editor del sistema operativo assiste l’utente nel creare nuovi programmi oppure nel modificare programmi esistenti. A tale scopo, visualizza sullo schermo i caratteri appena digitati consentendo così di rilevare eventuali errori di battitura; inoltre, consente all’utente di correggere parti di documento tramite operazioni di inserimento e cancellazioni di caratteri. Infine, l’editor si serve del sistema di archiviazione, sia per memorizzare in appositi archivi su disco nuovi programmi immessi dagli utenti, sia per reperire l’archivio contenente il programma su cui operare ulteriori modifiche.

Da vari anni sono ormai disponibili editor specializzati chiamati editor guidati dalla sintassi per mettere a punto programmi scritti in uno specifico linguaggio di programmazione. Tali editor si avvalgono della sintassi del linguaggio per riconoscere le parole chiave contenute nel programma, per evidenziarle tipograficamente e, soprattutto, per effettuare un riconoscimento automatico di eventuali errori di sintassi. Il loro uso migliora sensibilmente la produttività del programmatore.

### 3.3.5 Caricamento in memoria di un file eseguibile

Il caricamento di un file eseguibile in memoria RAM precede immediatamente la sua esecuzione e viene effettuato automaticamente dal sistema operativo in seguito alla richiesta di esecuzione del programma da parte dell’utente.

Le attività associate al caricamento dipendono dallo schema di gestione della memoria utilizzato e dal formato del file eseguibile. La Figura 3.4 illustra le fasi di messa a punto di un programma utilizzando le varie funzioni del sistema operativo appena descritte. Ritorneremo su questo argomento nel capitolo 8.

### 3.4 Supporto durante l’esecuzione

Si è discusso in precedenza delle varie fasi attraverso le quali passa un programma fino a iniziare l’esecuzione. Supponiamo ora che l’utente al terminale richieda
l’esecuzione di un programma. Inizia a questo punto da parte del sistema operativo, una fase di controllo del programma utente che si protrae fino alla sua terminazione.

Un primo tipo di controllo è quello relativo alla terminazione corretta del programma. Quando ciò si verifica, il NUCLEO del sistema operativo provvede ad inviare un apposito messaggio al processo di sistema associato al terminale (vedi capitolo 7) che può quindi rilasciare l’area di memoria e le altre risorse impegnate dal processo utente e prepararsi ad accettare il prossimo comando dell’utente.

Un altro tipo di controllo è quello relativo alla terminazione erronea del programma. Grazie ad esso, il sistema operativo è capace di riprendere il controllo anche in seguito a terminazioni anomale di un processo utente. Le terminazioni anomale controllabili sono quelle che inducono la CPU a generare un apposito segnale di interruzione quando si verificano. In effetti, esistono specifici segnali di interruzione in grado di rilevare i seguenti eventi:

- esecuzione da parte della CPU di una istruzione con codice operativo non valido;
- parametro dell’istruzione non valido (ad esempio, il programma cerca di effettuare un salto ad un indirizzo non valido);
• diritti d’accesso insufficienti (il programma cerca di indirizzare un’area protetta senza avere i privilegi necessari);

Quando il programma viene linkato in modo dinamico anziché statico, il sistema operativo deve provvedere un altro supporto fondamentale, ossia un linker dinamico. Come discusso in precedenza, il ruolo di tale supporto è di ritardare il linking delle funzioni di libreria richieste dal programma in esecuzione.

In alcuni casi, il traduttore prevede di linkare al programma utente un modulo di supporto all’esecuzione che effettua ulteriori controlli, oltre a quelli svolti dall’hardware del calcolatore. Il linguaggio Pascal, ad esempio, prevede un apposito programma preparato dal compilatore che coesiste in RAM insieme al programma dell’utente e che svolge varie funzioni, tra cui quella di filtrare ogni indirizzamento di variabili di tipo array per verificare se i valori degli indici rientrano nell’intervallo dichiarato. Nel caso opposto, il programma interrompe l’esecuzione del programma utente e prepara un messaggio di diagnostica specificando il nome dell’array e i valori degli indici al momento dell’errore.

Un altro prezioso strumento per la messa a punto di programmi è il debugger simbolico. Tale funzione richiamabile tramite appositi parametri in fase di compilazione e linkaggio, dà luogo al caricamento in memoria di un modulo aggiuntivo che interagisce con l’utente al terminale e serve a controllare l’esecuzione del programma. Mediante tale modulo è possibile, ad esempio, richiedere l’esecuzione del programma fino ad arrivare ad una frase specifica del programma sorgente e leggere quindi il valore attuale di alcune variabili. In tale ottica, gli interpreti si possono considerare come i moduli di controllo più completi poiché essi controllano ogni singola frase del programma anziché un limitato sottoinsieme. Come è già stato osservato, il prezzo da pagare per tale ricchezza di controlli è quello di avere tempi di esecuzione del programma interpretato molto più elevati di quelli del programma compilato.

### 3.5 Salvataggio/ripristino di dati

Anche se gli attuali file system sono alquanto affidabili, non si può escludere a priori il verificarsi di malfunzionamenti hardware dei dischi che potrebbero causare la perdita di tutte le informazioni in essi contenute. Per questo motivo, è consigliabile effettuare periodicamente il backup, ossia il salvataggio su supporti rimovibili del contenuto dei dischi del sistema. A tale scopo sono stati messi a punto programmi di utilità che consentono di svolgere tale attività in modo alquanto semplice. Le caratteristiche salienti di tali programmi sono le seguenti:
3.6 Amministrazione del sistema

Il termine *amministrazione del sistema* sta ad indicare una serie di funzioni attinenti all’uso corretto delle risorse del sistema multiprogrammato da parte degli utenti. In particolare, sono stati messi a punto programmi di utilità per risolvere i seguenti problemi:

- **Identificazione dell’utente**: Oggi la soluzione più adottata consiste nell’identificare ogni utente tramite un nome utente (username) ed una parola chiave (password). L’utente deve, per potere accedere alle risorse del sistema, identificarsi e digitare la propria parola chiave. Solo dopo che l’identificazione è stata portata a termine con successo, l’utente può accedere al sistema. E’ opportuno individuare un responsabile, l’amministratore del sistema, il quale è l’unico abilitato a creare nuovi utenti od a rimuovere utenti già esistenti.

- **Conteggio delle risorse utilizzate e relativo addebito**: In molte aziende i costi del sistema informativo sono distribuiti tra i reparti che usufruiscono del servizio. E’ quindi importante misurare l’uso da parte dei vari utenti delle principali risorse del sistema quali tempo di CPU, il prodotto del tempo per lo spazio su disco utilizzato, i volumi di stampa, ecc. Le statistiche raccolte sull’uso delle risorse sono anche molto utili per individuare comportamenti anomali nonché eventuali colli di bottiglia. Potrebbe risultare, ad esempio, dalle statistiche di utilizzo della stampante veloce che essa causa un ritardo nella terminazione dei job per via delle code di stampa sempre piene.

- **Autorizzazione alla condivisione delle risorse**: Risulta spesso necessario, nel caso di progetti che prevedono gruppi di programmatori ed utenti, prevedere una condivisione controllata delle cartelle e degli archivi di uso comune.

- salvataggio/ripristino su una serie di volumi rimovibili opportunamente etichettati mediante un apposito programma di utilità che prevede una apposita interfaccia per guidare l’utente nella fase di montaggio/smontaggio di volumi;

- possibilità di comprimere/decomprimere gli archivi riducendo così la quantità di memoria complessiva richiesta per il backup;

- possibilità di effettuare un backup selettivo basato sulla data dell’ultimo aggiornamento di ogni archivio. In tale modo vengono salvati periodicamente i soli archivi che sono stati modificati dopo l’ultimo backup effettuato.
Come nel caso della identificazione degli utenti, è opportuno concentrare nelle mani dell’amministratore del sistema anche le autorizzazioni o revoche di condivisione dell’informazione in modo da avere un responsabile unico per tale attività che può risultare alquanto critica.
Capitolo 4

IL FILE SYSTEM

4.1 Introduzione

Le memorie di massa, quali i dischi e i nastri magnetici, hanno caratteristiche diverse da quelle della memoria RAM.

In primo luogo, sono di tipo non volatile e consentono quindi di conservare informazioni registrate dagli utenti.

In secondo luogo, tali memorie hanno dei tempi di indirizzamento variabili che dipendono sia dall’indirizzo richiesto che dallo stato attuale del circuito di indirizzamento. Nel caso di un nastro magnetico, ad esempio, il tempo di indirizzamento dipenderà dalla distanza tra la porzione di nastro dove andrà eseguita l’operazione di I/O e quella attualmente posizionata dalla testina di lettura/scrittura. Nel caso di un disco a braccio mobile, invece, il tempo di indirizzamento dipenderà dalla distanza tra il cilindro contenente la traccia richiesta e quello attualmente posizionato dal braccio mobile dell’unità a disco. In conseguenza, diventa importante collocare l’informazione nelle memorie di massa in modo da ridurre, per quanto possibile, i tempi di indirizzamento.

In terzo luogo, infine, le informazioni contenute in tali memorie non sono direttamente accessibili alla CPU: a differenza della memoria principale, non esistono istruzioni che consentono alla CPU di trasferire informazioni tra registri interni e aree di disco. La stessa funzione è invece ottenuta utilizzando processori specializzati chiamati processori di I/O ed eseguendo le seguenti istruzioni:

1. istruzioni per preparare i parametri (indirizzo del disco, indirizzo della traccia, indirizzo del settore, numero di byte da trasferire, indirizzo in memoria)
da trasferire al processore di I/O al quale è collegato il disco. Nei calcolatori di tipo mainframe tali istruzioni prendono il nome di *programma di canale*;

2. istruzione di tipo *start i/o* per attivare il processore di I/O. Eseguita l’ultima istruzione, il programma si pone in attesa del segnale di interruzione di “fine I/O” da disco che indicherà il completamento dell’operazione.

Come si è appena visto, il reperimento e il trasferimento di informazioni contenute in memoria di massa risulta alquanto complesso. Nasce quindi l’esigenza di includere nei sistemi operativi appositi programmi chiamati sistemi di archiviazione (in inglese, *file system*) che rendano più agevole l’accesso alle informazioni contenute in tali memorie.

### 4.2 Caratteristiche dei file system

Il file system svolge i seguenti compiti:

- semplificare gli accessi al disco: grazie al file system, il programmatore che intende leggere o scrivere dati su disco non deve specificare gli indirizzi fisici dei settori da trasferire; il disco appare invece come un contenitore di file e directory e sono previste apposite API che consentono di creare, modificare e cancellare file e directory;

- consentire al programmatore di organizzare i suoi dati servendosi di una delle strutture logiche di dati previste ed offrire per ognuna gli operatori necessari ad operare su di essa\(^1\);

- gestire i vari dischi collegati al sistema, assegnando o rilasciando aree di memoria secondaria in base alle richieste dei programmi che utilizzano il file system.

Le cartelle (in inglese, *directory*) sono state introdotte per consentire agli utenti di raggruppare file tra loro affini nella stessa directory. Suddividendo i numerosi file contenuti in un disco in cartelle diverse le quali, a loro volta, possono includere altre cartelle oltre che file, risulta semplificato il reperimento dei singoli file e, come si vedrà più avanti, risulta possibile fare uso di schemi di protezione differenziati.

\(^1\)Linguaggi di programmazione quali Cobol e RPG includono apposite frasi per accedere a record di un file.
4.3 Protezione delle informazioni

Oltre a conservare nel tempo informazioni raggruppate in file, i file system offrono in generale alcuni schemi di protezione con finalità diverse. Per motivi di spazio, accenniamo soltanto a due di essi:

- gestione automatica di copie di riserva o backup. Ogni volta che viene modificato un file, il file system crea automaticamente un file contenente la versione precedente. Questo consente all’utente di non perdere i suoi dati nel caso di operazioni erronee. In alcuni file system, anziché una sola copia di backup viene creata una serie storica delle versioni precedenti indicizzata, appunto, sul numero di versione (versione 1 per il file appena creato, 2 per la prima modifica, ecc.). In tali file system viene lasciato all’utente il compito di cancellare le versioni considerate superflue.

- accesso riservato a cartelle o file. In un sistema multiprogrammato, i file di utenti diversi sono registrati in dischi comuni ed è quindi necessario garantire uno schema di protezione che impedisca ad utenti non autorizzati di accedere ai file di altri utenti. A tale scopo, si rivela molto utile il concetto di cartella poiché in tale modo risulta alquanto agevole raggruppare tutti i file di un utente in una stessa cartella facendo “vedere” all’utente la porzione di disco corrispondente alla sua cartella. In pratica, un simile approccio deve essere in qualche modo mitigato per consentire la condivisione controllata di informazioni. La cartella /tmp presente in tutti i filesystem Unix è un tipico esempio di cartella condivisa: qualunque utente può accedere ad essa per leggerne o modificarne il contenuto.

4.4 Il filesystem di Unix

L’intero sistema operativo Unix è basato su un file system che rimane tuttora valido, pur essendo stato progettato all’inizio degli anni ’70. È opportuno quindi iniziare la descrizione di Unix partendo da tale file system.

4.4.1 File

Un file Unix è un contenitore di informazioni strutturato come una sequenza di byte. Il NUCLEO Unix non interpreta il contenuto di un file. Sono disponibili
diverse librerie di programmi che realizzano astrazioni ad un livello più elevato: facendo uso dei programmi inclusi in tali librerie, è possibile ad esempio considerare un file come composto da una sequenza di record, ognuno dei quali è suddiviso in campi; è inoltre possibile caratterizzare uno o più campi di un record come campi chiave ed effettuare indirizzamenti veloci basati sull’uso di campi chiave. Tutti i programmi inclusi in tali librerie fanno uso tuttavia delle chiamate di sistema standard del NUCLEO Unix.

Dal punto di vista dell’utente, i file sono collocati in uno spazio dei nomi organizzato ad albero come indicato nella Figura 4.1.

Tutti i nodi dell’albero, tranne i nodi foglia, denotano nomi di cartelle. Ogni nodo di tipo cartella specifica quali nomi di file e quali nomi di cartelle sono contenuti in esso.

Un nome di file o directory consiste in una sequenza di caratteri ASCII, con l’eccezione del carattere ‘/’ e del carattere nullo ‘\0’ usato come terminatore di stringhe. Ogni file system Unix pone un limite alla lunghezza massima di un nome di file, tipicamente 255 caratteri.

La cartella associata alla radice dell’albero è chiamata cartella radice (in inglese, root directory). Per convenzione, il suo nome è rappresentato col carattere ‘/’. I nomi di file all’interno della stessa cartella devono essere distinti tra loro, anche se lo stesso nome di file può essere usato in cartelle diverse.

Unix associa una cartella attiva (in inglese, current working directory) ad ogni processo; essa fa parte del contesto di esecuzione del processo ed identifica la cartella attualmente usata dal processo. Quando, ad esempio, l’utente rossi effettua il login, la cartella attiva di tale utente è: /home/rossi; ogni esecuzione del comando cd modifica il valore della cartella attiva.

Per identificare un file, il programmatore fa uso di un nome di percorso (in inglese,
4.4. IL FILESYSTEM DI UNIX

pathname) che consiste in una alternanza di caratteri ‘/’ e di nomi di cartella. L’ultimo componente del nome di percorso può essere il nome di una cartella, oppure il nome di un file, a seconda che l’oggetto da identificare sia una cartella oppure un file.

Se il primo componente di un nome di percorso è uno ‘/’, allora il nome di percorso è di tipo assoluto, poiché la ricerca deve iniziare a partire dalla cartella radice. Nel caso opposto il nome di percorso è relativo poiché la ricerca inizia dalla cartella attiva del processo in esecuzione.

Nel costruire nomi di percorso, sono anche usati i simboli ‘.’ e ‘..’. Essi indicano, rispettivamente, la cartella attiva e la cartella che contiene la cartella attiva. Se la cartella attiva è la cartella radice, il significato dei simboli ‘.’ e ‘..’ coincide.

4.4.2 Link hard e soft

Ogni nome di file incluso in una cartella è chiamato hard link o più semplicemente link. Lo stesso file può avere più link inclusi nella stessa cartella, o in cartelle diverse, e quindi può avere più nomi.

Il comando Unix ln f1 f2 è usato per creare un nuovo hard link avente nome di percorso f2 relativo al file avente nome di percorso f1.

Gli hard link hanno due limiti:

- non è possibile creare un hard link relativo ad una cartella poiché questo potrebbe trasformare l’albero delle cartelle in un reticolo con cicli, rendendo con ciò impossibile la identificazione di un file in base al suo nome di percorso;

- un nuovo hard link f2 può essere creato soltanto nel file system che contiene il file f1; questo è un limite in quanto i sistemi Unix più recenti consentono di installare più file system di tipo diverso in più dischi o partizioni.

Citiamo due casi in cui risulta conveniente fare uso di hard link:

- L’utente teme di cancellare per errore un file contenente dati preziosi, per cui crea un secondo hard link per lo stesso file, possibilmente in una directory diversa. Nel caso di cancellazione erronea del file, potrà recuperare i dati dal secondo hard link.
L’utente intende confrontare tramite una applicazione ricorsiva del comando `diff` due directory simili di grandi dimensioni contenenti soltanto programmi sorgenti. Le due directory, pur contenendo un numero molto elevato di directory e file, differiscono per pochi file. Per questo motivo, nel creare la seconda directory, l’utente ha usato hard link ai corrispondenti file della prima directory nella maggior parte dei casi ed ha creato file appositi soltanto per i file che risultano diversi. Poiché il comando `diff` è in grado di verificare se i due file da confrontare sono gli stessi, il tempo richiesto per eseguire il `diff` ricorsivo sarà molto minore nel caso in cui siano stati creati molti hard link.

I limiti degli hard link sono superati facendo uso di soft link (sono anche chiamati link simbolici). Essi consistono in appositi file di dimensione ridotta che contengono il nome di percorso di un file. Tale nome di percorso è arbitrario; esso può identificare un file inserito in un qualsiasi file system, o perfino un file inesistente.

Il comando Unix `ln -s f1 f2` è usato per creare un nuovo soft link avente nome di percorso `f2` relativo al file avente nome di percorso `f1`. Quando questo comando viene eseguito, il file system crea un file di tipo link simbolico (vedi prossimo paragrafo) e scrive in esso il nome di percorso `f1`; inserisce quindi nella cartella opportuna una nuova voce contenente l’ultimo nome del nome di percorso `f2`. In questo modo, diventa possibile tradurre automaticamente ogni riferimento ad `f2` in un riferimento ad `f1`\(^2\).

È possibile leggere il valore di un soft link mediante il comando `ls` con l’opzione `-l`. Viceversa, lo stesso comando applicato ad un hard link elenca gli attributi del file.

Diamo due esempi d’uso dei soft link:

- I nomi delle librerie di programmi sono spesso espressi mediante soft link. Il link simbolico `libncurses.so`, ad esempio, punta al file contenente la versione di libreria installata, ossia `libncurses.so.5`. Questo accorgimento risulta utile nel realizzare programmi applicativi che non dipendono dalla specifica versione di libreria.

- L’utente A intende far vedere alcuni dei suoi programmi ad un altro utente, ad esempio l’utente B, senza però consentirgli di modificarli. A tale scopo crea in una directory di B un soft link per ogni programma che intende

\(^2\)Se il nome di percorso include dei softlink di directory che creano un ciclo, il sistema operativo si difende segnalando un errore dopo aver incontrato più di \(k\) (costante prefissata) softlink nel percorso.
4.4. IL FILESYSTEM DI UNIX

far vedere a B. Si osservi come ogni modifica fatta da A su uno di tali file risulta immediatevisibile a B.

4.4.3 Tipi di file

I file Unix sono divisi nelle seguenti classi:

- file regolari;
- cartelle;
- link simbolici;
- device file orientati a blocchi;
- device file orientati a caratteri;
- pipe con nome (spesso chiamata FIFO);
- socket

I primi tre tipi di file sono i componenti del file system Unix: l’informazione (dati o programmi) è contenuta nei file regolari; l’uso di cartelle consente di realizzare uno spazio dei nomi di file organizzato ad albero, e quindi di fare uso di nomi di percorso; i link simbolici, infine, offrono una maggiore flessibilità di gestione.

I device file identificano dispositivi di ingresso/uscita quali dischi, tastiera o mouse. Essi sono stati introdotti da Unix per realizzare uno standard di programmazione C in base al quale un gruppo ristretto di chiamate di sistema quali open(), close(), read() e write() può essere utilizzato per indirizzare dati contenuti in un file tradizionale o pure in un qualche dispositivo di ingresso/uscita.

Le pipe con nome o FIFO sono usate come un meccanismo veloce per la comunicazione tra processi. In pratica, il file di tipo pipe serve soltanto a memorizzare il nome della pipe, mentre lo scambio di dati tra processi avviene in RAM senza fare uso del file system.

I socket sono utilizzati sia come strumento di sincronizzazione locale che per la connessione in rete (vedi paragrafo 5.4.1).
4.4.4 Descrittore di file ed inode

Unix effettua una distinzione netta tra file e descrittore di file. Tutti i tipi di file tranne i device file e i socket sono considerati come contenitori di sequenze, eventualmente nulle, di caratteri. In altre parole, un file non include al suo interno alcuna informazione che ne descrive la tipologia o la struttura quale, ad esempio, la lunghezza del file oppure un delimitatore di tipo End-Of-File (EOF) che segnala la fine del file stesso.

Tutte le informazioni necessarie al file system per gestire un file sono contenute in un apposito descrittore chiamato inode, anch’esso registrato in modo permanente su disco. Ogni file possiede un proprio inode ed il file system usa il numero dell’inode come identificatore del file. In pratica, lo stesso file è identificato in due modi diversi: il programmatore si riferisce ad esso tramite un pathname corrispondente ad un qualche hard link; il filesystem identifica invece il file tramite l’indice del suo inode. Il filesystem è in grado di ricavare l’inode corrispondente ad pathname avvalendosi di informazioni contenute nelle directory incluse nel pathname.

Esistono svariate realizzazioni di file system per Unix che differiscono in modo notevole una dall’altra. In ogni caso, ognuna di esse deve rispettare lo standard POSIX che richiede la presenza dei seguenti attributi tra quelli che caratterizzano un inode:

- tipo di file type (vedi sopra);
- numero di hard link associati al file;
- lunghezza del file in byte;
- identificatore del dispositivo che include il file;
- numero dell’inode associato al file;
- User ID dell’utente possessore del file;
- Group ID del file;
- alcune date e ore che specificano, ad esempio, quando l’inode è stato indirizzato per ultimo e quando è stato modificato per ultimo;
- i diritti d’accesso ed il “file mode” (vedi appresso).
4.4. IL FILESYSTEM DI UNIX

4.4.5 Diritti d'accesso e file mode

I potenziali utenti di un file sono divisi in tre classi:

- l’utente possidente del file: quando un file è creato da un programma, lo User ID dell’utente per conto del quale tale programma viene eseguito viene usato per caratterizzare il possidente del file;

- gli utenti che appartengono allo stesso gruppo (che hanno lo stesso stesso group ID) dell’utente possidente del file;

- i rimanenti utenti (others).

Tre diversi tipi di diritti d’accesso chiamati Read (diritto di lettura), Write (diritto di scrittura) ed Execute (diritto di esecuzione del codice) sono previsti per ognuna di queste tre classi. L’insieme dei diritti d’accesso associati ad un file consiste quindi in nove indicatori (flag) binari che sono spesso indicati tramite 3 cifre nell’alfabeto ottale. Ad esempio, 755 significa che il possidente del file può leggere, scrivere ed eseguire il file, mentre gli utenti appartenenti al gruppo dell’utente possidente del file e gli altri utenti possono soltanto leggere ed eseguire il file.

Per essere eseguito un file deve avere sia il diritto di Read che quello di Execute.

Il significato dei diritti d’accesso per file di tipo directory è diverso: Read consente di leggere il contenuto della directory, Write consente di aggiungere o rimuovere file dalla cartella ed Execute consente di “attraversare” la directory, ossia di usare tale directory come componente di un nome di percorso.

Vi sono altri due indicatori chiamati suid (Set User ID) e sgid (Set Group ID) che consentono al possidente del file oppure agli utenti appartenenti al gruppo del possidente del file di eseguire il programma come superuser (root). Il comando `passwd` è un classico esempio di `programma suid`: l’utente non ha diritto a leggere la tabella contenente le password ma il superuser ha tale diritto. In effetti, lanciando il comando `ls -l /usr/bin/passwd` si ottiene in risposta: `-rws--x--x`. La lettera `r` denota il fatto che il programma è di tipo suid.

Un ultimo flag chiamato sticky non è più usato.

Complessivamente i diritti d’accesso ed il file mode richiedono 12 bit che sono spesso codificati mediante 4 cifre ottali. Ad esempio, 0644 significa `suid=0, sgid=0, sticky=0, rw-, r--, r--`. 
4.4.6 API per la gestione di file

Esistono due tipi di funzioni di libreria per gestire file:

- Le funzioni di “stream I/O” definite dallo standard ANSI C.
- Le API per la gestione di file definite dallo standard POSIX.

Le API dello standard C fanno riferimento a puntatori ad oggetti di tipo FILE che devono essere dichiarati nel programma mediante frasi del tipo:

```c
FILE *f1, *f2;
```

Le API usate dallo standard POSIX fanno riferimento a indici di tipo numero intero chiamati file descriptor. Tali indici devono essere dichiarati nel programma mediante frasi del tipo:

```c
int fd1, fd2;
```

Le API dello standard C prevedono, nel caso di file regolari, uno schema di bufferizzazione automatica: i dati in lettura possono essere letti in anticipo in buffer gestiti dalle API. In modo analogo dati in scrittura possono essere accumulati in buffer prima di essere scritti su disco.

Le API usate dallo standard POSIX sono delle semplici funzioni C di interfaccia: ognuna di esse si limita ad invocare la chiamata di sistema dopo aver passato i parametri di tale chiamata nel modo opportuno (vedi paragrafo 9.6).

La tabella 4.1 illustra alcune corrispondenze tra i due tipi di API.

Sappiamo dal capitolo 1 che lo standard ANSI C assicura la portabilità dei programmi in qualunque sistema dotato di un compilatore ANSI C, mentre lo standard POSIX garantisce soltanto la portabilità in sistemi di tipo “POSIX compliant”. Poiché il nostro obiettivo è quello di studiare sistemi operativi di tipo Unix, discuteremo ed useremo, vuoi nelle dispense, vuoi nei programmi proposti nelle esercitazioni, le API di tipo POSIX `open()`, `close()`, `read()`, `write()` ed `lseek()`. Useremo tuttavia nei programmi le API ANSI C `printf()` e `scanf()` per effettuare in modo agevole conversione di tipi.
4.4. IL FILESYSTEM DI UNIX

<table>
<thead>
<tr>
<th>API ANSI C</th>
<th>API POSIX</th>
<th>descrizione</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen()</td>
<td>open()</td>
<td>apertura file</td>
</tr>
<tr>
<td>fclose()</td>
<td>close()</td>
<td>chiusura file</td>
</tr>
<tr>
<td>fread()</td>
<td>read()</td>
<td>lettura</td>
</tr>
<tr>
<td>fwrite()</td>
<td>write()</td>
<td>scrittura</td>
</tr>
<tr>
<td>fseek()</td>
<td>lseek()</td>
<td>posizionamento all’interno del file</td>
</tr>
<tr>
<td>printf()</td>
<td></td>
<td>conversione di variabili seguita da stampa</td>
</tr>
<tr>
<td>scanf()</td>
<td></td>
<td>lettura da tastiera seguita da conversione</td>
</tr>
</tbody>
</table>

Tabella 4.1: Corrispondenza tra API ANSI C e API POSIX.

4.4.7 API POSIX per la gestione di directory

Le directory sono file speciali strutturati in record a lunghezza variabile composti da tre campi: lunghezza in byte del nome del file, nome del file e numero dell’inode.

Per motivi di sicurezza, le directory non sono trattate come file regolari ma possono essere consultate e modificate soltanto tramite apposite API POSIX (il linguaggio C ignora l’esistenza di directory).

La tabella 4.2 elenca alcune di tali API.

<table>
<thead>
<tr>
<th>API</th>
<th>descrizione</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir()</td>
<td>creazione directory</td>
</tr>
<tr>
<td>rmdir()</td>
<td>eliminazione directory</td>
</tr>
<tr>
<td>opendir()</td>
<td>apertura directory</td>
</tr>
<tr>
<td>closedir()</td>
<td>chiusura directory</td>
</tr>
<tr>
<td>readdir()</td>
<td>lettura elemento directory</td>
</tr>
<tr>
<td>chdir()</td>
<td>impostazione nuova present working directory</td>
</tr>
<tr>
<td>getcwd()</td>
<td>lettura present working directory</td>
</tr>
</tbody>
</table>

Tabella 4.2: API POSIX per la gestione di directory.

4.4.8 Chiamate di sistema per la gestione di file

Quando un utente indirizza il contenuto di un regular file, oppure quello di una cartella, egli indirizza in pratica dei dati registrati in un block device, ossia in un dispositivo di ingresso/uscita di tipo disco. In questo senso, un file system può
essere considerato come una astrazione a livello utente della organizzazione fisica di una qualche partizione di disco. Poiché l’utente non è abilitato, per motivi di sicurezza, ad interagire direttamente col disco, ogni operazione attinente ad un file deve essere svolta in Kernel Mode.

Per questo motivo, Unix prevede un insieme di chiamate di sistema per la gestione dei file; quando un processo intende interagire con uno specifico file, esso invoca la chiamata di sistema opportuna passando il nome di percorso del file come parametro. Esaminiamo brevemente tali chiamate di sistema.

### 4.4.9 Apertura di un file

Per potere indirizzare un file, esso deve essere stato “aperto” in precedenza. A tale scopo, il processo esegue la seguente chiamata di sistema:

\[
fd = \text{open(path, flag, mode)}
\]

I tre parametri usati hanno il seguente significato:

- **path**: Nome del percorso (assoluto o relativo) del file da aprire.

- **flag**: Specifica come il file deve essere aperto: solo lettura, solo scrittura, lettura/scrittura, scrittura in fondo al file (append); specifica anche, nel caso in cui non esista alcun file avente nome di percorso path, se esso deve essere automaticamente creato.

- **mode**: Specifica i diritti d’accesso (nel solo caso in cui il file venga creato).

Il NUCLEO gestisce tale chiamata di sistema creando un oggetto “file aperto” e restituendo al processo un identificatore chiamato “descrittore di file”. Un oggetto file aperto contiene:

- alcune strutture di dati per la gestione del file, quali un puntatore all’area di memoria in cui verranno copiati i caratteri del file, un campo offset che indica l’indice all’interno del file dell’ultimo carattere indirizzato e così via;

- alcuni puntatori a funzioni (i metodi dell’oggetto) che il processo è autorizzato ad invocare; tale insieme dipende dal valore del parametro flag.
Diamo ora alcune ulteriori informazioni relative all’interazione tra file e processo che si evincono dalla semantica POSIX.

Un descrittore di file identifica l’interazione tra un processo ed un file aperto, mentre un oggetto “file aperto” contiene dati relativi a tale interazione.

Più processi possono aprire lo stesso file in modo concorrente: in tale caso, il file system assegna ad ognuno di essi un apposito descrittore di file ed un apposito oggetto “file aperto”. Quando ciò si verifica, il file system di Unix non tenta in alcun modo di sincronizzazione le diverse operazioni di ingresso/uscita eseguite da più processi sullo stesso file. È comunque disponibile la chiamata di sistema flock() che consente a più processi di sincronizzarsi sull’intero file o su una porzione di esso. Per creare un nuovo file, è anche possibile fare uso della API create() che è gestita dal NUCLEO alla stregua di una open().

### 4.4.10 Indirizzamento di un file

Dopo averlo aperto in precedenza, è possibile indirizzare un file regolare Unix in modo sequenziale oppure casuale (random); i character device file e le FIFO sono solitamente indirizzati in modo sequenziale. In entrambi i tipi di accesso, il NUCLEO aggiorna il valore del campo “offset” dell’oggetto file aperto in modo che esso punti al prossimo carattere da leggere o da scrivere all’interno del file.

Il file system assume di norma che il nuovo indirizzamento è sequenziale rispetto a quello effettuato per ultimo: le chiamate di sistema read() e write() fanno sempre riferimento all’attuale valore del campo offset dell’oggetto “file aperto”. Per modificare tale valore, è necessario invocare in modo esplicito la chiamata di sistema lseek().

Quando un file viene aperto, il NUCLEO imposta il campo offset alla posizione del primo byte nel file, ossia pone offset uguale a 0. La chiamata di sistema lseek() opera sui seguenti parametri:

\[
\text{newoffset} = \text{lseek}(\text{fd}, \text{offset}, \text{whence});
\]

che hanno i seguenti significati:

- **fd**: Descrittore di file del file aperto;
- **offset**: Specifica un intero usato per calcolare il nuovo valore di offset;
• whence: Specifica se newoffset, il nuovo valore del campo offset, deve essere calcolato come il valore di offset (offset dall’inizio del file, macro SEEK_SET), la somma di offset più l’attuale valore del campo offset (offset dalla posizione attuale, macro SEEK_CUR), la differenza tra l’indice dell’ultimo byte nel file e offset (offset dalla fine del file, macro SEEK_END).

La chiamata di sistema read() fa uso dei seguenti parametri:

\[ nread = \text{read}(fd, buf, count); \]

che hanno i seguenti significati:

• \textit{fd}: Descrittore di file del file aperto che si intende leggere;
• \textit{buf}: Indirizzo dell’area di memoria nello spazio degli indirizzi del processo dove trasferire i dati letti;
• \textit{count}: Numero di byte da leggere.

Nel gestire tale chiamata di sistema il NUCLEO cerca di leggere count byte dal file identificato da \textit{fd}, a partire dal byte individuato dal campo offset nel descrittore dell’oggetto “file aperto”. In alcuni casi (fine del file, pipe vuota, ecc.), il NUCLEO non riesce a leggere tutti i byte richiesti.

Il valore \textit{nread} restituito dalla chiamata di sistema specifica il numero di byte effettivamente letti. Il nuovo valore del campo offset è ottenuto sommando il valore precedente a \textit{nread}. I parametri della chiamata di sistema \textit{write()} sono simili a quelli della \textit{read()}.

Il NUCLEO esegue le chiamate di sistema \textit{read()}, \textit{write()} e \textit{lseek()} in modo atomico: due chiamate di sistema non possono essere eseguite simultaneamente sullo stesso file.

4.4.11 Chiusura di un file

Quando un processo non ha più necessità di indirizzare un file, esso invoca la chiamata di sistema:

\[ \text{res} = \text{close}(fd); \]
che rilascia l’oggetto “file aperto” relativo al descrittore di file `fd`. Quando un processo termina, il NUCLEO chiude tutti i file ancora aperti del processo.

### 4.4.12 Cambiamento di nome e cancellazione di file

Per potere cambiare nome oppure cancellare un file, un processo non deve aprire il file. In effetti, tali operazioni non agiscono sul contenuto del file bensì sul contenuto di una o più cartelle dove inserire o rimuovere il nome del file. La chiamata di sistema:

```c
res = rename(oldpath, newpath);
```

cambia il nome di un link al file, mentre la chiamata di sistema:

```c
res = unlink(pathname);
```

cancella il file specificato da `pathname` dalla parent directory e decrementa di uno il contatore di link al file. Il file viene cancellato dal file system quando tale contatore assume il valore 0.
Capitolo 5

INTERFACCIA CON L’UTENTE

5.1 Introduzione

In un sistema operativo dedicato, ogni utente collegato al sistema deve essere in grado, servendosi dei diversi dispositivi di ingresso previsti, di specificare in modo agevole quale tra i vari servizi riconosciuti ed eseguibili dal sistema egli intende ottenere. Per soddisfare questo requisito, il sistema operativo include appositi programmi per riconoscere i segnali di ingresso inviati dall’utente; tali programmi costituiscono complessivamente una interfaccia tra l’utente da una parte, e il resto del sistema operativo e l’hardware dall’altra.

Le interfacce dei sistemi operativi possono avere caratteristiche alquanto diverse, anche se il loro compito rimane sempre quello di identificare la richiesta effettuata dall’utente nonché gli eventuali parametri ad essa associati. Si possono distinguere tre approcci possibili basati, rispettivamente, sull’uso di menu, comandi ed icone.

5.2 Interfaccia a menu

Nelle interfacce basate sull’uso di menu, l’utente non deve ricordare i nomi dei diversi programmi che intende eseguire poiché ogni menu che appare sullo schermo include un elenco di funzioni (o classi di funzioni) eseguibili e, per ognuna di esse, è fornita una breve descrizione nonché il nome del tasto da premere per
richiederne l’esecuzione. In generale, data la limitata capacità dello schermo, è necessario ricorrere a una gerarchia di menu per descrivere tutte le funzioni offerte dal sistema operativo. Il pregio di tali interfacce, tuttora utilizzate in sistemi transazionali ed all’interno di vari programmi applicativi, sta nella loro semplicità: in primo luogo, esse non richiedono l’uso di un terminale grafico ma possono operare anche su semplici terminali alfanumerici. In secondo luogo, i programmi di gestione di tali interfacce sono alquanto semplici e compatti. Le interfacce a menu si rivelano tuttavia troppo rudimentali per consentire ad un utente di interagire con un moderno sistema operativo.

Anche le interfacce realizzate tramite terminali “touch screen” si possono considerare come interfacce a menu. L’utente identifica sullo schermo una icona e preme il dito su di essa. In corrispondenza, appare un’altra schermata con altre icone. Dopo un certo numero di interazioni, compare sullo schermo l’informazione richiesta. I terminali touch screen consentono anche di rappresentare sullo schermo una tastiera associando una apposita area sensibile ad ogni tasto.

5.3 Interfaccia a comandi

Tale interfaccia consente all’utente di interagire col sistema tramite una serie di comandi. Ogni comando è costituito da una sequenza di caratteri alfanumerici seguita dal tasto di <INVIO>. I comandi hanno una loro sintassi ben definita che è usata da un modulo del sistema operativo chiamato interprete di comandi; tale modulo analizza ogni comando ed esegue le azioni richieste. Tipicamente il comando inizia col nome seguito da una serie di parametri ed opzioni.

In realtà, è più corretto parlare negli attuali sistemi operativi di linguaggi di comandi anziché di semplici comandi. In effetti, è possibile creare file contenenti sequenze di comandi: quando l’interprete di comandi riceve il nome di tale file come programma da eseguire, passa ad interpretare i vari comandi contenuti nel file.

I linguaggi di comandi prevedono l’uso di variabili locali, variabili di sistema e frasi condizionali, per cui rappresentano, in diversi casi, una valida alternativa ai linguaggi compilati. I programmi scritti in un linguaggio di comandi prendono il nome di script.

Al solito, la scelta se realizzare una applicazione facendo uso di uno script oppure di un programma compilato (ad esempio, dal compilatore C) dipende dai tempi di risposta richiesti: l’esecuzione di un script risulta ovviamente più lenta poiché ogni comando è interpretato dall’interprete di comandi. D’altra parte, i sistemi
operativi includono solitamente librerie molto ampie di comandi già predisposti che possono essere opportunamente combinati in uno script, per cui il tempo di messa a punto dell’applicazione risulta molto inferiore quando si ricorre ad uno script.

Le interfacce a comandi continuano ad essere le interfacce preferite dai programmatori professionisti per la loro estrema flessibilità d’uso, anche se possono risultare poco gradite ad un utente occasionale il quale non realizza perché sia necessario fare uso di comandi alquanto complessi per svolgere semplici funzioni.

5.3.1 Lo shell di Unix

L’interfaccia tra l’utente ed il sistema operativo Unix è una interfaccia a comandi. I comandi Unix sono alquanto potenti per due motivi diversi:

- Non esiste un insieme predefinito di comandi: ogni programma eseguibile o interpretabile tramite uno degli interpreti inclusi nel sistema operativo è considerato come un comando. L’utente può quindi creare comandi personalizzati e richiederne l’esecuzione semplicemente digitando il nome del nuovo comando seguito dagli opportuni parametri e premendo il tasto di <INVIO>.

- Unix consente di creare comandi composti da comandi più semplici. A tale scopo, Unix include un apposito linguaggio di comandi che si distingue da altri simili linguaggi per la sua ricchezza e flessibilità. Esso è un vero e proprio linguaggio di programmazione dotato di frasi di controllo e chiamate di procedura. A differenza dei linguaggi di programmazione convenzionali quali il C oppure il FORTRAN, i comandi scritti in tale linguaggio non devono essere compilati da un apposito compilatore ma vengono immediatamente letti, interpretati ed eseguiti da un interprete di comandi chiamato shell (guscio). Esistono diversi shell (C shell, Bourne shell, bash shell, ecc.) che differiscono leggermente l’uno dall’altro. Viene inoltre offerta all’utente la possibilità di definire il proprio shell da usare in applicazioni particolari.

Vediamo ora quali sono le caratteristiche più interessanti di tale linguaggio di comandi che, secondo una consuetudine invalsa, chiameremo shell.

Un comando Unix è un frase composta dal nome del comando seguito eventualmente da parametri separati da uno spazio. Dopo aver digitato il comando, è necessario digitare il tasto di <INVIO> per indicare il completamento del comando stesso. Nell’eseguire, ad esempio, il comando:
cap file1 file2

lo shell effettua una copia del file chiamato file1 ed assegna a tale copia il nome file2 (se esiste già un altro file avente tale nome, esso viene distrutto e sostituito dal nuovo).

I caratteri immessi da tastiera e quelli visualizzati sullo schermo del terminale sono considerati dallo shell come due file speciali chiamati, rispettivamente, input e output. In assenza di altre informazioni, lo shell assume che il file stdin (file di ingresso) sia associato alla tastiera e quello stdout (file di uscita) sia associato al video. Nell’eseguire, ad esempio, il comando:

date

lo shell visualizza la data e l’ora attuale sul video.

La ridefinizione dell’input e dell’output è una delle varie caratteristiche che rendono lo shell molto flessibile. Usando gli operatori “>” per ridefinire l’output e “<” per ridefinire l’input, è possibile specificare file di ingresso e di uscita diversi da quelli standard. Il comando:

date > file1

scrive, ad esempio la data e l’ora attuale nel file file1 anziché sul video.

Servendosi di tale linguaggio, è possibile specificare mediante frasi di controllo del tipo if then else e while, l’ordine mediante il quale eseguire la sequenza di comandi contenuta nel programma e controllare tale esecuzione in funzione dei parametri ricevuti e delle condizioni di terminazione dei vari comandi eseguiti.

Daremo diversi esempi di script Unix in apposite esercitazioni. Suggeriamo comunque allo studente di provare il maggior numero possibile di comandi tra quelli documentati tramite il comando man.

Avvertenza: dopo avere creato un file di comandi, ossia uno script, dovete assegnare al file il diritto di esecuzione con un comando del tipo:

chmod +x my_script
5.3. INTERFACCIA A COMANDI

5.3.2 Un esempio di script bash

Il seguente script elenca tutti i file del tipo *.c e *.h che contengono almeno una occorrenza di una stringa a partire da una directory prefissata (vedi anche script cerca_stringa incluso nella esercitazione 3):

```bash
#!/bin/bash
#
# il primo parametro denota la directory, il secondo la stringa da cercare
#
if [ ! ${ECHO} ]; then
  ECHO=/bin/echo
fi
if [ ! ${FIND} ]; then
  FIND=/usr/bin/find
fi
if [ ! ${GREP} ]; then
  GREP=/usr/bin/grep
fi
if [ ! ${ROOTSEARCH} ]; then
  ROOTSEARCH=$1
fi
cd ${ROOTSEARCH}
${ECHO} Ricerca di "$2" a partire da "${ROOTSEARCH}\"
${FIND} . -name "*.\[ch]\" -exec ${GREP} -l "$2" {} \;
${ECHO} Ricerca terminata.
```

5.3.3 Struttura dello shell

L’interprete di comandi o shell è un programma memorizzato in un apposito file eseguibile del file system. Durante l’inizializzazione del sistema, Unix lancia uno shell per ognuno dei terminali installati; ognuno di tali shell visualizza un messaggio di “login” sul proprio terminale e rimane in attesa fino a quando l’utente non inizia una sessione comunicando il suo nome e la relativa parola d’ordine. Da quel momento, tutti i file creati da tale terminale avranno come possessore lo User ID (UID) dell’utente attualmente "loggato".

Se il login risulta corretto, lo shell responsabile del terminale si pone in attesa di comandi digitati dall’utente. Quando l’utente invia un nuovo comando, lo shell
CAPITOLO 5. INTERFACCIA CON L’UTENTE

primario crea uno shell secondario per eseguire il programma corrispondente. Se il comando non deve essere eseguito in background, lo shell primario aspetta la terminazione dello shell secondario. Se, viceversa, l’utente ha richiesto di eseguire un comando in background (il nome del comando è seguito dal carattere ‘&’), lo shell primario ritorna immediatamente attivo.

La distinzione tra shell principale e shell secondari consente non soltanto di avviare programmi in background ma anche di realizzare annidamenti di shell: ciò consente di realizzare degli script alquanto sofisticati.

Vediamo ora come lo shell associa al nome di un comando il file contenente il programma o lo script da eseguire. In teoria, si potrebbe richiedere di digitare ogni volta l’intero pathname ma tale soluzione non sarebbe gradita dall’utente in quanto lo costringerebbe a digitare lunghe sequenze di caratteri, ad esempio /usr/bin/man anziché man. Il problema viene risolto introducendo una apposita variabile shell chiamata PATH che contiene la sequenza delle directory, separate dal carattere ‘:’ dove cercare il file avente un nome prescelto.

Il contenuto di PATH può essere personalizzato inserendo una nuova definizione, diversa da quella di default, nel file .profile contenuto nella home directory dell’utente.

Possiamo usare il comando:

    echo $PATH

per visualizzare sullo schermo il valore della variabile PATH, ossia i nomi di tutte le directory che verranno esaminate dallo shell per cercare un file il cui nome sia uguale a quello del comando. In risposta a tale comando, appare sul monitor una sequenza di pathname del tipo:

    /usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/sbin:.

Se il comando digitato dall’utente non può essere eseguito, lo shell segnala una condizione di errore. Questo avviene per due possibili motivi:

- il file system non contiene un file avente il nome del comando;
- il file avente il nome del comando non è né un file eseguibile né un file di comandi per il quale sia disponibile un opportuno interprete (tale tipo di file è chiamato comando indiretto).
5.4. **INTERFACCE GRAFICHE**

Lo shell è in grado di distinguere un file eseguibile da un file contenente comandi da interpretare leggendo i primi byte del file: essi consentono di identificare il programma che deve essere usato per eseguire il file (loader ELF o interprete di un qualche tipo). Nel caso di script interpretabili dallo shell `bash`, essi devono iniziare\(^1\) con la stringa:

```
#!/bin/bash
```

Non esiste infatti un solo interprete: esistono diversi interpreti per gli shell Unix ed esistono inoltre altri interpreti per i vari linguaggi interpretati riconosciuti dal sistema operativo (ad esempio, Java).

Come è ovvio, se i primi byte del file non consentono allo shell di identificare il programma che deve essere usato per eseguire il file, esso invia un messaggio di errore del tipo:

```
command not found
```

Concludiamo questa presentazione sommaria degli shell Unix con una osservazione importante. Oltre ad essere digitati da utenti al terminale, i comandi possono essere lanciati da un qualsiasi programma in esecuzione tramite l'API `system()` oppure tramite una chiamata di sistema del tipo `execve()` (vedi Capitolo 8).

Grazie a tali API risulta quindi sempre possibile scrivere un programma che simula la sequenza di azioni svolte da un utente al terminale.

### 5.4 Interfacce grafiche

Le interfacce basate sull’uso di icone sono state introdotte per rispondere a particolari esigenze di facilità d’uso e di apprendimento e sono quindi rivolte in primo luogo ad utenti non necessariamente esperti. A differenza dei due tipi di interfacce descritte in precedenza che richiedono soltanto un terminale alfanumerico ed una tastiera, le interfacce grafiche richiedono invece l’uso di un terminale grafico, di una tastiera e di un apposito dispositivo d’ingresso chiamato mouse mediante il quale è possibile spostare il cursore grafico sullo schermo ed inviare segnali al sistema premendo uno dei pulsanti collocati in cima ad esso. L’idea di base è quella di offrire all’utente un menu grafico composto da immagini chiamati *icone*.

\(^1\)Ciò non è necessario se lo shell `bash` è lo shell di default.
Il primo sistema operativo commerciale con interfaccia grafica è stato messo a punto per i personal computer Macintosh della Apple negli anni '80. In tale sistema, il menu iniziale includeva una serie di icone corrispondenti ai file dell’utente, a quelli contenenti programmi del sistema operativo e ad altri file di servizio come la “cartella vuota” e il “cestino”. Per cancellare, ad esempio, un file, ad esempio il file di nome A, in tale sistema è sufficiente:

1. posizionare il cursore dentro all’icona A (questo si ottiene spostando il mouse sulla scrivania finché la freccia del cursore non sia contenuta nell’icona);
2. spostare, tenendo premuto il pulsante del mouse, l’icona A in quella denominata “cestino” rilasciando quindi il pulsante.

Per aprire un documento gestito da una apposita applicazione, è sufficiente effettuare un doppio clic sull’icona corrispondente.

In alcuni casi, però, l’uso della tastiera rimane fondamentale: ogni qualvolta si intende creare una nuova cartella, oppure cambiare il nome di un file, è necessario trasmettere l’informazione al file system facendo uso della tastiera. Oggi le interfacce grafiche sono diventate uno standard per i sistemi operativi dei personal computer e sono molto diffuse nei sistemi operativi MacOs della Apple, in quelli di tipo Unix ed in quelli della famiglia Microsoft Windows.

5.4.1 X Window

A differenza di altri sistemi operativi quali MacOS della Apple, oppure Windows XP della Microsoft, che incorporano l’interfaccia grafica all’interno del NUCLEO, l’interfaccia grafica per Unix è sempre realizzata tramite un programma applicativo. In effetti, Unix è nato prima che esistessero i terminali grafici necessari per realizzare tale tipo di interfaccia.

Il più noto programma applicativo Unix che realizza una interfaccia grafica si chiama X Window. Esso è stato sviluppato inizialmente presso il Massachusetts Institute of Technology (MIT) ed è oggi lo standard de facto di interfaccia grafica per sistemi Unix.

X Window è una applicazione client/server. Al server X Window competono le seguenti attività:

- Gestire il mouse e la tastiera. Se l’input si riferisce alla proprietà di una finestra (scelta finestra attiva, trascinamento o ridimensionamento finestra,
5.4. **INTERFACCE GRAFICHE**

ecc.), effettuare i cambiamenti richiesti dall’utente. Se l’input si riferisce ad una applicazione (nuovo carattere per un editor, ecc.), passare tale input al client X Window opportuno.

- Disegnare sul monitor all’interno delle finestre riservate agli X client i dati che tali client hanno inviato al server.
- Trascinare, ingrandire o rimpicciolire finestre in base alle indicazioni fornite dal mouse.

Ai client X Window competono invece le seguenti attività:

- Aspettare di ricevere dati dal server per poi elaborarli.
- Inviare al server i risultati dell’elaborazione.

La comunicazione tra server e client avviene tramite socket. Il client interagisce con il server facendo uso delle apposite API del “socket programming”. Un tipico schema di programma per interagire con il server è il seguente:

```c
socket(...); /* creazione di un socket */
connect(...); /* connessione col server */
....
read(...);
.... /* scambio di dati tra client e server */
write(...);
....
close(...); /* fine della connessione */
```

Questo consente, disponendo di una rete sufficientemente veloce, di collegarsi ad un calcolatore remoto e di usare il server locale per interagire con client remoti. In altre parole, risulta possibile disponendo di un “network filesystem” quale NFS, visualizzare sul proprio schermo le icone che rappresentano il file system di una macchina remota.

Poiché X Window è stato concepito come una applicazione client/server basata su socket, tale estensione non richiede in pratica alcuna riprogrammazione di X Window.

Sono stati sviluppati numerosi client specializzati per X Window, ad esempio per emulare terminali alfanumerici (**xterm**), per visualizzare documentazione Unix
all’interno di una finestra (xman), per visualizzare un orologio (xclock), ecc. Inoltre, moltissimi programmi applicativi operano oggi in ambiente X Window, ossia chiedono al server di creare una propria finestra e comunicano con esso in accordo col protocollo X Window ricevendo ed inviando dati.

5.5 Realizzazione di interfacce grafiche in Unix

X Window è in qualche modo il motore usato nei sistemi di tipo Unix per offrire una interfaccia grafica agli utenti. Anche se è possibile in teoria realizzare un nuovo client usando le numerose funzioni offerte da X Window, gli sviluppatori preferiscono avvalersi di apposite librerie che rendono molto più agevole la programmazione. La libreria Qt, ad esempio, è realizzata nel linguaggio C++ ed offre funzioni molto potenti che consentono di creare nuove finestre, gestire i click del mouse all’interno di esse, disegnare pulsanti ed associare funzioni ai click su tali pulsanti, ecc.
Capitolo 6

NUCLEO E PROCESSI

6.1 Introduzione

Pur tenendo conto della continua diminuzione del costo dell’hardware, i processori di tipo CPU continuano ad essere considerati risorse hardware pregiate il cui uso deve essere ottimizzato. Per questo motivo, gli attuali sistemi operativi sfruttano il potenziale parallelismo hardware dell’elaboratore avviando e controllando l’esecuzione di più programmi o sequenze di istruzioni sui vari processori inclusi nell’elaboratore. Viene in tale modo aumentata la produttività o throughput del sistema, ossia il numero di programmi eseguiti per unità di tempo. Per conseguire tale risultato, è necessario fare uso di tecniche di gestione alquanto sofisticate.

L’idea di base comune a tali sistemi è quella di caricare simultaneamente in memoria più programmi e di eseguirne alcuni in parallelo, utilizzando i diversi processori a disposizione. Tale impostazione è nota come multitasking.

È interessante osservare come sia possibile realizzare sistemi multitasking con ottime prestazioni, anche per calcolatori dotati di due soli processori: una CPU e un processore di I/O. In effetti, l’esecuzione di un programma è costituita da una successione di fasi di elaborazione sulla CPU e fasi di attesa per il completamento di operazioni di I/O che impegnano il processore di I/O lasciando logicamente inattiva la CPU. Durante tali fasi di attesa, il sistema operativo multitasking può mandare in esecuzione sulla CPU altri programmi tra quelli caricati in memoria, migliorando così la produttività del sistema.

La Figura 6.1 illustra in modo semplificato l’aumento di produttività derivante dalla esecuzione concorrente di tre programmi, chiamati A, B e C, rispetto alla esecuzione sequenziale in ambiente uniprogrammato.
CAPITOLO 6. NUCLEO E PROCESSI

Come si può osservare dalla figura, l’esecuzione multitasking di più programmi distinti consente, in generale, di aumentare sensibilmente la produttività del sistema, ossia il numero di programmi eseguiti per unità di tempo. Oggi la gestione multitasking è utilizzata in tutti i tipi di sistemi operativi (gestione a lotti, sistemi interattivi, sistemi transazionali, controllo di processi). Vedremo in un capitolo successivo come i diversi obiettivi che tali sistemi devono conseguire si rifletta in diverse tecniche di gestione dei processori.

Da un punto di vista funzionale, si distinguono i seguenti tipi di parallelismo:

- **multitasking**: esecuzione di programmi indipendenti sulla CPU e sul processore di I/O;
- **multiprogrammazione**: multitasking con l’aggiunta di tecniche di protezione della memoria che assicurano che un programma in esecuzione non possa accedere alle aree di memoria assegnate agli altri programmi eseguiti insieme ad esso;
- **multiprocessing** multiprogrammazione estesa ad elaboratori dotati di più CPU e processori di I/O.

Si noti come il multitasking non implichi la multi-utenza, ossia l’uso simultaneo del sistema da parte di più utenti. In effetti, le prime versioni di MacOs della Apple o Windows 95 della Microsoft erano sistemi multitasking che consentivano l’esecuzione di più programmi indipendenti a beneficio di un unico uten-
6.2. ARCHITETTURA E MODELLO DI ESECUZIONE

te. Oggi, i principali sistemi operativi per personal computer sono tutti sistemi multiprogrammati che consentono il collegamento simultaneo di più utenti.

6.2 Architettura e modello di esecuzione

Da alcuni decenni ormai, i calcolatori, anche quelli del tipo personal computer contengono più processori che condividono una memoria comune e devono pertanto essere considerati come sistemi concorrenti in grado di svolgere più attività indipendenti allo stesso tempo. L’architettura hardware a cui si farà riferimento è rappresentata schematicamente in Figura 6.2.

Come si può osservare, esistono almeno due processori autonomi, ossia una CPU e un processore di ingresso/uscita (il nome usato per tale tipo di processore dipende dal tipo di calcolatore. Nei calcolatori del tipo personal computer esso viene chiamato DMA (un acronimo per Direct Memory Access), mentre nei mainframe prende il nome di processore di I/O o canale di I/O. Nei sistemi mainframe o server aventi elevate prestazioni, sono solitamente presenti più CPU e più processori di I/O.

Qualunque sia il tipo di calcolatore, i diversi processori del sistema accedono ad una memoria comune rispettando un protocollo che consente ad essi di emettere richieste per ottenere e per rilasciare il bus del sistema a cui è collegata la RAM.

Poiché tale protocollo è realizzato interamente in hardware, ogni processore può essere programmato come se fosse l’unico a accedere alle informazioni contenute in memoria.

Esiste quindi nei sistemi concorrenti un parallelismo potenziale di operazioni a livello hardware, nel senso che è possibile fare in modo che processori distinti eseguano allo stesso tempo istruzioni appartenenti a programmi diversi.
Vediamo ora come deve essere organizzato un sistema operativo per sfruttare al meglio i vari processori inclusi nell’elaboratore.

### 6.3 Processi

La nozione di *processo* (in inglese process o task) è emersa all’inizio degli anni ’60 insieme allo sviluppo dei primi sistemi multitasking.

In passato, quando i sistemi operativi erano ancora molto primitivi, una volta caricato in RAM un programma, la CPU era impegnata al 100% nell’eseguire le istruzioni del programma e l’esecuzione terminava soltanto quando veniva eseguita l’apposita istruzione di *halt*.

Oggi, i compiti di un sistema operativo multitasking si sono ampliati notevolmente e lo scenario è diventato più complesso. In tali sistemi, infatti, sono caricati in RAM diversi programmi tra loro indipendenti, per cui risulta necessario distinguere l’attività svolta da una CPU dalla esecuzione di un programma: infatti, in un intervallo di tempo prefissato, una CPU può alternativamente eseguire sequenze di istruzioni appartenenti a programmi diversi mentre, durante lo stesso intervallo, l’esecuzione di un programma su quel processore o su altri può essere ripresa e sospesa più volte.

Per potere controllare l’esecuzione dei vari programmi caricati in RAM, il sistema operativo associa ad ognuno di tali programmi un apposito descrittore chiamato *processo*.

Il descrittore di tipo processo è composto da diverse strutture di dati che contengono svariate informazioni riguardo alle varie risorse attualmente ottenute dal programma in esecuzione. Riprenderemo questo tema nel capitolo 9 ma possiamo già citare due informazioni indispensabili incluse in ogni processo:

- Gli indirizzi fisici in RAM (vedi sezione ??) in cui è caricato il programma. Tali indirizzi sono una risorsa posseduta dal processo che non può essere assegnata ad altri processi, tranne nel caso in cui il programma condivida una parte della sua memoria con altri programmi.

- Il contenuto dei registri della CPU al momento in cui sta per iniziare l’esecuzione del programma, oppure al momento in cui è stata sospesa l’esecuzione

\footnote{Vedremo più avanti nel capitolo 7 che l’esecuzione di programmi che fanno uso di apposite API forza il sistema operativo a creare più di un processo in corrispondenza ad uno singolo programma.}
6.3. **PROCESSI**

del programma per l’ultima volta. Tale informazione consente al sistema operativo di iniziare o continuare l’esecuzione del programma.

Il descrittore di tipo processo è indispensabile per realizzare sistemi multitasking poiché consente al sistema operativo sia di sospendere l’esecuzione di un programma per un tempo indeterminato che di riprendere l’esecuzione di un programma a partire dal punto in cui essa era stato sospesa.

I processi sono spesso (e impropriamente) considerati come entità autonome. In effetti, il processo è soltanto un descrittore di risorse usato dal sistema operativo e non esegue alcuna azione; useremo però nel resto di questo capitolo il termine “processo” anziché “programma associato al processo” per motivi di semplicità.

Definiremo quindi la vita di un processo come l’intervallo di tempo che inizia quando il programma associato al processo è caricato in RAM e termina quando termina l’esecuzione del programma.

Definiremo inoltre come _avanzamento del processo_ l’esecuzione da parte di una CPU di istruzioni incluse nel programma associato al processo. Quando un processo non è in esecuzione su alcuna CPU, esso non esegue istruzioni del programma e quindi non avanza. Vice versa, quando il processo è in esecuzione, una CPU è usata per fare avanzare il processo. Il tempo complessivo di CPU usato dal processo è una misura grossolana dell’avanzamento di un processo.

Si noti come la definizione di programma da noi usata include sia il codice che i dati su cui opera il programma, per cui programmi che eseguono lo stesso codice operando su dati diversi sono considerati programmi diversi. Se, ad esempio, si lanciano più compilazioni C in background su file sorgenti diversi il codice comune ai vari processi di compilazione è il compilatore C ma ogni compilazione opera su dati diversi.

Facendo ricorso al concetto di processo, possiamo affermare che il compito del sistema operativo è quello di controllare l’avanzamento di un gruppo di processi con il duplice obiettivo di soddisfare le richieste degli utenti e di mantenere impegnati i diversi processori.

Per meglio capire come questo possa essere realizzato, si considera l’avanzamento di un processo come una serie di transizioni da uno stato all’altro e si distinguono tre stati di un processo chiamati: _esecuzione, pronto e bloccato_.

- _in esecuzione_: il processo è in esecuzione quando un processore sta eseguendo istruzioni del programma associato al processo;
• **pronto**: il processo è in attesa di un processore in grado di eseguire istruzioni del programma;

• **bloccato**: il processo è bloccato in attesa che si verifichi uno specifico evento. Quando ciò avviene, il sistema operativo cambia lo stato del processo da bloccato a pronto.

Gli stati appena indicati sono una schematizzazione che, nella pratica, deve essere espansa notevolmente, come appare dalle seguenti considerazioni:

• Nei sistemi multiprocessore, vi è un processo in esecuzione su ognuna delle \( N \) CPU installate ed il sistema operativo deve sapere su quale CPU un processo è in esecuzione; per questo motivo, si usa uno stato di esecuzione distinto per ogni CPU.

• Il componente del sistema operativo che sceglie, tra i vari processi in stato di pronto, quello da mettere in esecuzione è chiamata *scheduler* e verrà discusso alla fine di questo capitolo. Per svolgere il suo compito, lo scheduler suddivide lo stato di pronto in vari stati, ognuno dei quali esprime una diversa “priorità” dei processi in quel particolare stato.

• Passando allo stato bloccato, vi sono numerosi motivi, tra loro diversi, per cui un processo rimane in tale stato; ogni specifico cambiamento di stato del sistema potrà servire, di norma, a sbloccare solo alcuni tra i vari processi bloccati. Lo stato bloccato deve quindi essere decomposto in tanti stati quanti sono i tipi di eventi che possono bloccare i processi.

### 6.4 Il NUCLEO coordinatore di processi

Il componente del sistema operativo che coordina l’avanzamento dei processi e notifica ad essi il verificarsi di eventi prende il nome di *NUCLEO* (in inglese, *kernel*). Chiaramente, il NUCLEO non è un processo esso stesso ma costituisce la parte più critica dell’intero sistema operativo. Come vedremo in un successivo capitolo, le parti meno critiche del sistema operativo possono essere realizzate tramite appositi processi mentre particolare attenzione va posta nel progettare un NUCLEO affidabile ed efficiente.

La comunicazione tra un processo ed il NUCLEO avviene tramite un insieme predefinito di funzioni che prendono il nome di *chiamate di sistema* (in inglese, *system call*).
L’insieme di chiamate di sistema disponibili costituisce in un certo senso il biglietto da visita del sistema operativo: ogni servizio aggiuntivo offerto da un sistema operativo, ad esempio la lettura degli attributi di un file oppure la creazione di un nuovo processo, dà luogo ad apposite chiamate di sistema. Tali funzioni devono essere considerate come delle porte di accesso al NUCLEO: un processo non può richiedere di eseguire un generico programma del NUCLEO né tantomeno indirizzare variabili del NUCLEO. Esso può soltanto richiedere di eseguire una delle chiamate di sistema esistenti. Facendo ricorso alla terminologia della programmazione ad oggetti, il NUCLEO può essere considerato come un oggetto specifico che offre svariate (alcune centinaia) funzioni di interfaccia, ossia le chiamate di sistema. L’implementazione del NUCLEO, ossia le funzioni e strutture di dati utilizzate, rimane nascosta al programmatore.

Spesso, viene introdotto un ulteriore strato di software tra i programmi degli utenti e quelli del NUCLEO: si tratta di funzioni di tipo API (Application Programming Interface) contenute in apposite librerie di programmi (vedi paragrafo 1.3). Tali funzioni che rendono più agevole la richiesta di servizi al NUCLEO includono sempre una o più chiamate di sistema.

La Figura 6.3 illustra le interazioni tra programma utente, API, chiamata di sistema e NUCLEO: in tale figura, si fa riferimento alla API malloc() usata dai programmatori C per ottenere memoria dinamica. Tale funzione non è una chiamata di sistema bensì una API che utilizza la chiamata di sistema brk() per ottenere memoria dinamica dal NUCLEO.

In conclusione, il ruolo della libreria API è essenzialmente quello di facilitare la richiesta di servizi al NUCLEO: le API possono essere considerate come delle funzioni di interfaccia ad alto livello che risultano più facili da utilizzare delle chiamate di sistema.
6.5 Risorse usate dai processi

Procedendo nella presentazione del modello, si osserva che i processi, come gruppo di entità autonome, non sono sufficienti a rappresentare il funzionamento di un sistema multitasking: in effetti, non si può escludere, anzi è un caso frequente, che due o più processi richiedano simultaneamente l’uso dello stesso processore, oppure che essi richiedano di accedere allo stesso dispositivo di I/O (disco, scheda di rete, stampante, ecc.).

Per consentire di trattare in maniera uniforme l’assegnazione ai processi di componenti hardware e software del sistema, si introduce il concetto di risorsa come qualunque oggetto che i processi usano e che condiziona il loro avanzamento. In base a tale definizione, l’avanzamento di un processo è condizionato dalla disponibilità di risorse di vario tipo. Uno dei compiti tipici dei sistemi operativi multitasking è quello di controllare l’uso di risorse da parte dei processi in modo da risolvere eventuali conflitti.

Prima di descrivere, nel prossimo paragrafo, come ciò possa essere realizzato, si introducono alcuni attributi importanti che servono a meglio caratterizzare le varie risorse gestite. A tale scopo, si classificano le risorse in:

- risorsa permanente: può essere utilizzata ripetutamente da più processi senza che il suo stato venga modificato.
- risorsa consumabile: è una risorsa transiente gestita dal sistema operativo dall’istante in cui è creata a quello in cui è eliminata. Essa può essere creata sia da un processo che dal NUCLEO; in modo analogo, può essere eliminata sia da un processo che dal NUCLEO.

Le interazioni tra un processo $P$ e una risorsa permanente $R$ avvengono secondo il seguente schema:

1. richiesta di $R$ da parte di $P$;
2. assegnazione da parte del NUCLEO di $R$ a $P$;
3. utilizzo di $R$ da parte di $P$;
4. rilascio di $R$ da parte di $P$.

Una risorsa permanente è seriale se può essere utilizzata da un solo processo alla volta, è condivisa (shared) nel caso opposto.
6.6. **SCHEDULING DI PROCESSI**

Una tabella di dati che può essere indirizzata da un solo processo alla volta è una risorsa permanente seriale. Viceversa, una tabella di dati che può essere usata da più processi allo stesso tempo, ad esempio una libreria di programmi, è una risorsa permanente condivisa.

Un classico esempio di uso di risorse consumabili è il seguente: si è visto in precedenza che un processo $Q$ può essere bloccato in attesa di un evento di tipo $E$. Si supponga che un altro processo $P$ sia quello le cui azioni daranno luogo al verificarsi di tale evento. In tale caso, $P$ è il produttore della risorsa consumabile $E$ mentre $Q$ è il consumatore. Inoltre, si dirà che $Q$ è *sincronizzato* rispetto a $P$, nel senso che rimarrà bloccato fino a quando $P$ non avrà eseguito una determinata azione.

Parleremo più diffusamente di tali risorse quando affronteremo il problema della sincronizzazione tra processi.

### 6.6 Scheduling di processi

Come detto in precedenza, un compito importante del NUCLEO è quello di distribuire l’uso della CPU in modo equo tra i vari processi in stato di pronto.

Il problema più generale trattato in questo paragrafo è quello di determinare in che ordine debbano essere soddisfatte le richieste dei processi per risorse di vario tipo. La soluzione di tale problema, chiamata *scheduling*, è essenzialmente una strategia che, tenendo conto di diversi obiettivi, consente al sistema operativo di scegliere il prossimo processo a cui assegnare una delle risorse gestite, non appena essa si rende disponibile.

Poiché tali risorse hanno caratteristiche tra loro molto diverse, un sistema operativo include, in generale, più funzioni di scheduling. Per semplicità, si concentrerà l’attenzione in questo paragrafo su due di esse chiamate *scheduling a breve termine* e *scheduling a lungo termine*.

Tornando alla nozione di scheduling, è importante osservare che, in assenza di obiettivi espliciti, qualunque strategia ha lo stesso valore, per cui potrebbe risultare soddisfacente in tale contesto usare funzioni di scheduling che determinano in modo del tutto casuale l’ordine secondo il quale soddisfare le richieste dei processi.

In realtà, questo non avviene poiché ogni sistema operativo è progettato per conseguire, sia pure con enfasi diverse, due obiettivi principali: la prevedibilità dei tempi d’esecuzione dei processi e l’utilizzazione delle risorse.
6.6.1 Prevedibilità dei tempi d’esecuzione

Tale obiettivo si riferisce al fatto che, sia nei sistemi batch che in quelli interattivi, è importante garantire all’utente un livello di servizio minimo, qualunque sia l’occupazione del sistema al momento in cui è eseguito il job o il comando.

Tale *livello di servizio* può essere definito in vari modi: nei sistemi interattivi, si misura generalmente come tempo di risposta del sistema nell’eseguire un semplice comando, ad esempio un comando di editing. Nei sistemi a lotti (batch), si usano invece misure più complesse come il *tasso di servizio*, ossia il numero di unità di servizio ricevute dal job per unità di tempo. Tali unità di servizio tengono conto sia del tempo di CPU che del numero di KByte x ora (area di memoria RAM per unità di tempo) ottenute dal job da quando è stato immesso nel sistema.

6.6.2 Utilizzazione delle risorse

Questo obiettivo è considerato prioritario nei sistemi di grandi dimensioni e, in particolare, nei sistemi di gestione a lotti. Data la diversità delle risorse gestite, il sistema operativo stabilisce una gerarchia di priorità: le risorse più pregiate sono i processori, seguiti dalla memoria, dalla memoria secondaria e, infine, dalle unità a nastro e a disco che consentono di montare volumi rimovibili e dalle stampanti.

È utile sottolineare che mentre è relativamente agevole utilizzare in modo ottimale una singola risorsa, risulta invece impossibile, date le caratteristiche dei job eseguiti in multiprogrammazione, determinare uno scheduling che ottimizzi tutte le risorse allo stesso tempo. Una buona utilizzazione delle risorse, ottenibile tramite una messa a punto (tuning) dei parametri associati alle varie funzioni di scheduling, consente di aumentare la produttività del sistema e quindi, indirettamente, di migliorare la prevedibilità dei tempi di risposta a parità di carico di lavoro.

6.6.3 Scheduling a breve termine

Lo *scheduling a breve termine* è utilizzato per gestire le risorse hardware più pregiate del sistema, ossia le CPU. Per gli altri processori, ossia per i canali di I/O, non è necessario effettuare scheduling in quanto tali risorse sono assegnate in modo seriale ai processi dall’inizio del trasferimento dati alla fine. In altre parole, non è possibile sospendere per poi riprendere più tardi un programma di canale che realizza un trasferimento dati.
 Quando un processo è in esecuzione su una CPU, esso può cambiare stato con conseguente invocazione della funzione di scheduling, nei seguenti casi:

- il processo in esecuzione $P$ passa nello stato di bloccato, per cui rilascia la CPU che può essere riassegnata ad un qualche altro processo;
- $P$ termina eseguendo una _exit()_ (vedi paragrafo 7.5.1);
- $P$ rilascia volontariamente la CPU eseguendo _sched_yield()_ (vedi paragrafo 7.3);
- il NUCLEO verifica che $P$ ha esaurito il suo quanto di tempo (vedi paragrafo 2.4.3 e prossimo paragrafo);
- un processo $P'$ diverso da $P$ passa dallo stato di bloccato a quello di pronto. Questo cambiamento di stato è causato da un qualche segnale di interruzione che è gestito da un programma del NUCLEO. Dopo avere registrato il cambiamento di stato di $P'$, il NUCLEO invoca la funzione di scheduling: se la priorità di $P'$ è maggiore di quella di $P$ esso mette in esecuzione $P'$.

In pratica, lo scheduling a breve termine è eseguito con frequenza elevata poiché i suddetti cambiamenti di stato di processi si verificano decine o centinaia di volte al secondo. Per questi motivi, la funzione è interamente realizzata da programmi del NUCLEO e non può essere svolta da un processo.

Si illustrano brevemente due tipiche strategie valide, rispettivamente, per sistemi interattivi e per quelli con gestione a lotti.

**Priorità decrescenti e multiplamento nel tempo della CPU**

Si associa ad ogni processo una priorità dinamica che può essere aggiornata durante il suo avanzamento: un processo interattivo inizia con la priorità massima $prior_1$; se utilizza un tempo $T_1$ di CPU senza terminare, la sua priorità diventa $prior_2 < prior_1$; se utilizza un tempo $T_1 + T_2$ di CPU senza terminare, la sua priorità diventa $prior_3$; e così via fino a raggiungere una priorità minima $prior_k$.

Lo stato di pronto è decomposto in tante code quante sono le priorità (vedi Figura 6.4).

Nello scegliere un nuovo processo da porre in esecuzione, il NUCLEO considera per primo la coda dei processi aventi priorità massima; se tale coda è vuota, considera quella dei processi aventi priorità immediatamente inferiore a quella
CAPITOLO 6. NUCLEO E PROCESSI

Figura 6.4: Code di priorità tra i processi in stato di pronto.

Figura 6.5: Time sharing della CPU.

massima, e così via. Quando un processo ha una determinata priorità $p_i$, il tempo di CPU $T_i$ non è concesso al processo in un unico blocco ma in un gruppo di $n_i$ quanti di tempo di durata $Q_i$ ($n_i \times Q_i = T_i$), secondo la tecnica della partizione di tempo (time-sharing) già introdotta nel primo capitolo.

Tale tecnica, illustrata tramite un semplice esempio in Figura 6.5, consiste nell’assegnare a turno un quanto $Q_i$ di CPU ad ogni processo con priorità $p_i$; quando un processo termina il suo quanto, esso viene posto in fondo alla coda opportuna.

La motivazione dietro tale strategia è duplice: privilegiare i processi che richiedono poco tempo di CPU e garantire ad essi un tasso di avanzamento uniforme grazie al multiplamento nel tempo della CPU.
Priorità variabili in funzione dell’uso dei processori

La strategia usata da sistemi per la gestione a lotti (batch) è quella di ottimizzare la produttività del sistema multiprogrammando gruppi di processi che impegnano i diversi processori del sistema. A tale scopo, il sistema operativo misura ogni T unità di tempo le percentuali di utilizzo dei processori da parte di ognuno dei processi utenti multiprogrammati. In base a tali rilevamenti, è possibile distinguere i processi che, durante l’ultimo intervallo di tempo misurato, hanno utilizzato maggiormente la CPU da quelli che hanno utilizzato maggiormente i processori di I/O. In seguito all’esame effettuato, la funzione di scheduling diminuisce le priorità dei processi del primo tipo e aumenta quelle dei processi del secondo tipo. La giustificazione per tale euristica è che, impegnando maggiormente i processori di I/O e quindi i dispositivi di I/O in generale, aumenterà la produttività del sistema.

6.6.4 Scheduling a lungo termine

Tale tipo di scheduling avviene solo nei sistemi operativi di gestione a lotti dotati di un batch monitor in grado di controllare l’esecuzione dei vari job nel sistema (vedi paragrafo 2.4.2). In tali sistemi, i vari job da eseguire hanno due importanti caratteristiche:

- l’utente dichiara tramite apposite richieste nella scheda iniziale del job e/o nelle schede iniziali dei vari step le risorse richieste dall’intero job, oppure dal singolo step;
- le richieste si riferiscono spesso a risorse che richiedono un lungo periodo di inizializzazione prima di poter essere utilizzate (risorse con elevato tempo di setup). Molti job richiedono, ad esempio, di montare volumi di dischi o nastri prefissati, oppure di inserire sulla stampante carta di tipo speciale, ecc. In ognuno di questi esempi è richiesto l’intervento dell’operatore per montare o smontare volumi o per cambiare carta e la durata di tali interventi è dell’ordine dei minuti.

Le funzioni che realizzano lo scheduling a lungo termine sono eseguite solo quando inizia o termina un lavoro (job) di un utente.

Distinguiamo due tipi di situazioni:

- lo scheduling di job per quanto riguarda l’uso di unità periferiche dedicate
CAPITOLO 6. NUCLEO E PROCESSI

(stampanti, unità a nastro, plotter, ecc.) è solitamente effettuato dall’operatore del sistema: usando opportuni comandi privilegiati della console del sistema, egli può assegnare unità periferiche a un job e iniziare l’esecuzione.

- lo scheduling di altre risorse, ad esempio la memoria principale o lo spazio su disco, è effettuato da processi di sistema (processi specializzati che svolgono funzioni del sistema operativo, vedi ultimo capitolo) chiamati INITIATOR: essi corrispondono ai processi shell in un sistema interattivo e cercano di ottenere tutte le risorse necessarie per eseguire uno step di un job.

In alcuni casi, può essere effettuata la preemption o prerilascio di uno o più step di un job per consentire ad un altro step di un nuovo job di andare in esecuzione immediatamente.

Se, ad esempio, l’INITIATOR di un nuovo job che dà luogo a un processo con priorità massima non trova la memoria necessaria per caricare il programma richiesto, il NUCLEO può richiedere il prerilascio da parte di altri processi meno prioritari delle aree di memoria ad essi assegnate. In tale caso, il NUCLEO dovrà salvare su disco tutte le aree di memoria usate dai processi preemptati in modo da essere in grado di rimettere successivamente tali processi in esecuzione.
Capitolo 7

PROGRAMMAZIONE CONCORRENTE

7.1 Introduzione

Rispettando l’approccio “top down” seguito finora, introduciamo il concetto di programma concorrente ed illustriamo il supporto offerto dal sistema operativo per realizzare programmi di tale tipo. Rimandiamo invece ad un capitolo successivo la descrizione delle principali azioni svolte dal NUCLEO nel trattare le chiamate di sistema attinenti alla programmazione concorrente.

I programmi si possono classificare in sequenziali e concorrenti a seconda del numero di processi utilizzati per eseguirli: nel primo caso, l’esecuzione di un programma dà luogo ad un singolo processo; nel secondo caso, dà luogo a più processi che competono tra loro per l’uso delle risorse del sistema.

Da un punto di vista funzionale, non vi è differenza tra i due tipi di programmazione: tutto quello che può essere realizzato da un programma concorrente può anche essere realizzato da un programma sequenziale. Dal punto di vista della produttività, invece, i programmi concorrenti sono, in generale, più efficienti di quelli sequenziali, specialmente quando eseguiti su macchine dotate di più CPU.

Esistono numerose applicazioni, ad esempio nel campo del calcolo numerico ed in quello della elaborazione delle immagini, in cui l’algoritmo utilizzato si presta ad essere codificato come un gruppo di programmi autonomi che possono essere eseguiti indipendentemente uno dall’altro. Anche se più complicata della programmazione sequenziale, la programmazione concorrente consente di realizzare programmi che hanno tempi di esecuzione più ridotti rispetto a quelli dei program-
CAPITOLO 7. PROGRAMMAZIONE CONCORRENTE

La programmazione concorrente è diventata ancora più cruciale con l’apparizione di chip “multicore”, ossia chip dotati di due o più processori. In effetti, l’unico modo per utilizzare i vari processori all’interno del chip consiste nel mandare in esecuzione il maggior numero possibile di programmi concorrenti.

Oltre a dare luogo a più processi, i programmi concorrenti devono fare in modo che i vari processi che cooperano nell’eseguire il programma si sincronizzino tra loro in modo corretto. Inoltre, nella maggior parte dei casi, i vari processi non operano su dati disgiunti per cui il sistema operativo deve offrire ai processi del gruppo la possibilità di operare su dati comuni.

Esamineremo i due problemi separatamente iniziando col presentare una classificazione dei principali schemi di interazione tra processi.

Concludendo questa breve introduzione, appare evidente come ogni sistema operativo che supporta la programmazione concorrente deve offrire al programmatore strumenti che consentono di:

- creare nuovi processi;
- eliminare processi esistenti;
- imporre vincoli di sincronizzazione tra processi concorrenti;
- condividere dati tra i processi concorrenti.

7.2 Schemi di sincronizzazione

Curiosamente, anche se vi è un numero praticamente infinito di programmi concorrenti diversi tra loro, gli schemi di sincronizzazione tra i vari processi possono essere ricondotti a tre tipi diversi che indicheremo col nome di:

- attesa;
- mutua esclusione;
- produttore/consumatore.
7.2. SCHEMI DI SINCRONIZZAZIONE

7.2.1 Attesa

Questo è lo schema alla base di tutti gli schemi di sincronizzazione. In altre parole, per risolvere un qualunque problema del tipo mutua esclusione o produttore/consumatore è necessario potere consentire ai processi di bloccarsi finché un determinato evento non si verifica.

Come vedremo più avanti in questo capitolo, vi sono numerosi tipi di attesa possibile. Per concretezza, accenniamo ad una di esse, ossia l’attesa del tipo “terminazione processo”: ogni volta che il processo padre crea un gruppo di processi figli per eseguire in parallelo una qualche elaborazione, esso dovrà sincronizzarsi mediante tale tipo di attesa con i processi figli prima di potere segnalare che l’elaborazione è stata completata.

Abbiamo già incontrato nel paragrafo 5.3.3 un esempio di schema di sincronizzazione di questo tipo: l’interprete di comandi crea un nuovo processo per mandare in esecuzione il file eseguibile specificato dall’utente, per poi porsi in attesa della terminazione di tale processo. Lo shell principale riprenderà la sua esecuzione soltanto dopo che lo shell secondario avrà segnalato in qualche modo di avere svolto il compito assegnatoli. Questo è ovviamente una attesa del tipo “terminazione processo”.

7.2.2 Mutua esclusione

Il problema della mutua esclusione di una risorsa contesa da più processi è quello di garantire che, ad ogni istante, vi sia al più un processo che occupi la risorsa e che ogni processo richiedente ottenga l’uso della risorsa entro un intervallo limitato di tempo.

La mutua esclusione si applica sia a risorse hardware che a risorse software.

Nel primo caso, poiché le risorse harware sono solitamente gestite dal sistema operativo, tocca a quest’ultimo risolvere il problema di mutua esclusione. Se, ad esempio, un processo ha ottenuto l’uso della stampante per stampare un file, il sistema operativo deve garantire che nessun altro processo potrà usare la stampante finché il primo processo non abbia terminato di stampare il file.

Nel caso di risorse software, il problema di mutua esclusione deve spesso essere risolto dal programmatore. Se, ad esempio, un programma concorrente prevede che più processi accedano simultaneamente a record di un file, il programmatore deve assicurare che la scrittura di un record di dati nel file sia serializzata in modo
da evitare che altri processi leggano lo stesso record quando l’aggiornamento non è stato ancora completato.

Esistono varianti al problema della mutua esclusione: in alcuni casi, i processi che richiedono la risorsa possiedono ognuno una priorità, ossia un numero che esprime il tipo di privilegio che essi hanno rispetto alla risorsa. Quando si libera una risorsa, essa è assegnata al processo con priorità massima e, nel caso ve ne sia più di uno, a quello che ha atteso più a lungo.

Un’altra variante prevede, oltre alle priorità, l’uso del prerilascio (in inglese, pre-emption) della risorsa per soddisfare immediatamente le richieste di processi prioritari. In questo caso, non appena giunge una richiesta da parte di un processo avente priorità sufficientemente elevata, il NUCLEO sospende il processo che usa attualmente la risorsa ponendolo in stato di pronto ed assegna quindi la risorsa al nuovo processo.

In realtà, la preemption richiede una qualche forma di cooperazione da parte del processo candidato alla preemption. Per evitare che ciò si verifichi in punti critici che comprometterebbero l’integrità di qualche struttura di dati, il programma deve includere opportuni punti di sincronizzazione che indicano i punti in cui il corrispondente processo può essere preemptato in tutta sicurezza. Il NUCLEO effettua la preemption soltanto quando il processo ha raggiunto un qualche punto di sincronizzazione.

### 7.2.3 Produttore/consumatore

Questo terzo schema di interazione è alla base di tutti gli scambi di informazione tra processi tra loro interagenti. Si distingue il processo produttore di informazioni da quello consumatore. Si noti come, a differenza della mutua esclusione che si applica anche a processi logicamente indipendenti (processi operanti su dati diversi), lo schema del produttore/consumatore implica che due o più processi si scambino informazioni, ossia che operino su dati comuni.

Illustriamo alcune importanti varianti del problema che schematizzano problemi di sincronizzazione comuni ai sistemi operativi multitasking.

- **un produttore/un consumatore**: l’unico produttore $P$ produce in istanti imprevedibili informazioni che devono essere raccolte e trattate dall’unico consumatore $Q$. $Q$ non può procedere se il buffer da cui prelevare dati è vuoto; viceversa, $P$ non può produrre se non vi è un buffer libero in cui depositare il nuovo gruppo di byte.
7.2. SCHEMI DI SINCRONIZZAZIONE

- **più produttori/un consumatore**: la funzione di spooling di uscita descritta nel paragrafo 2.4.2 è un programma concorrente che prevede una interazione di tipo produttore/consumatore tra i vari processi utenti che producono file di stampa e il processo OUTPUT WRITER che li consuma trasferendoli sulla stampante.

- **più produttori/più consumatori**: per aumentare le prestazioni del programma concorrente (posto che esistano le opportune risorse hardware) può essere opportuno dedicare un gruppo di processi per svolgere in multiprogrammazione, e quindi con maggiore efficienza, la stessa funzione. Nei sistemi per la gestione a lotti, ad esempio, i lavori sono suddivisi in classi in base alla priorità e alle risorse richieste. Ad ogni classe è associato almeno un processo del sistema operativo chiamato INITIATOR; tale processo esamina le richieste del lavoro e dei suoi passi e provvede a soddisfarle, oltre che a controllare l’esecuzione di ogni passo. Allo stesso tempo, è realizzato uno spooling di entrata che consente a più unità di input locali o remote di inviare simultaneamente lavori. A tale scopo, ogni unità di input è gestita da un apposito processo di sistema chiamato INPUT READER; tale processo smista i lavori ricevuti, copiandoli in file diversi a seconda della loro classe e notifica quindi uno degli INITIATOR associati alla classe del lavoro appena ricevuto.

**Il ruolo dei buffer**

Abbiamo accennato all’inizio del paragrafo precedente al fatto che il produttore deve disporre di un buffer libero in cui copiare i nuovi dati. Quanti buffer assegnare per realizzare uno schema di interazione produttore consumatore?

Se sia il tasso di produzione $t_p$ (misurato in byte al secondo) che quello di consumo $t_c$ sono costanti e $t_p \leq t_c$, un solo buffer è sufficiente. Consideriamo, ad esempio, il processo produttore che legge un DVD ed deposita in dati nel buffer. Il lettore DVD produce pacchetti a tasso costante perché la velocità di rotazione del disco è costante per cui il tasso $t_p$ del produttore è costante. Viceversa, il processo consumatore che elabora tali pacchetti e produce le immagini sul video nonché l’audio ha un tasso di consumo $t_c$ variabile in quanto possono verificarsi dei picchi di attività nel sistema durante i quali il processo consumatore è schedulato meno frequentemente. In assenza di buffer, vi sarà una perdita di pacchetti e la qualità del video e dell’audio si deterioreranno.

Se viceversa si usano $N > 1$ buffer, si riduce la probabilità di perdere pacchetti. Ovviamente, l’aggiunta di ulteriori buffer non porterebbe alcun beneficio qualora il tasso medio $t_c$ fosse minore del tasso medio $t_p$. 
7.3 Primitive di sincronizzazione

Per risolvere problemi di sincronizzazione del tipo “mutua esclusione” e “produttore/consumatore” è necessario disporre di apposite funzioni chiamate primitive di sincronizzazione.

Ogni sistema operativo possiede le proprie primitive di sincronizzazione. Inoltre, esistono numerose varianti, sia per quanto riguarda i nomi, sia per quanto riguarda la definizione delle chiamate di sistema che riguardano la sincronizzazione tra processi. In questo paragrafo ci limitiamo a descrivere sommariamente le caratteristiche di tali primitive senza preoccuparci di come siano realizzate. In un paragrafo successivo esamineremo in dettaglio le primitive di sincronizzazione offerte dai sistemi operativi di tipo Unix.

- **lock() / unlock()**: sono usate per risolvere problemi di mutua esclusione su una qualunque risorsa. Quando un processo $P$ esegue $\text{lock}(\text{lck})$, il NUCLEO verifica se la variabile $\text{lck}$ chiamata $\text{lock}$ ha il valore 1 (lock aperto). Nel caso affermativo, il NUCLEO assegna a $\text{lck}$ il valore 0 (lock chiuso) e rimette $P$ in esecuzione. Se $\text{lck}$ vale 0, il NUCLEO blocca $P$ fino a quando $\text{lck}$ non diventa uguale a 1. Quando un processo $P$ esegue $\text{unlock}(\text{lck})$, il NUCLEO verifica se esiste almeno un processo bloccato su $\text{lck}$, nel caso affermativo, sblocca uno di essi; altrimenti, assegna a $\text{lck}$ il valore 1. In entrambi i casi, il NUCLEO rimette $P$ in esecuzione.

La serializzazione di una risorsa permanente $R$ contesa da più processi si ottiene associando a $R$ un lock $\text{lck}_R$ e facendo in modo che ogni processo esegua $\text{lock}(\text{lck}_R)$ prima di utilizzare $R$ e $\text{unlock}(\text{lck}_R)$ dopo averla utilizzata.

- **$P()$ e $V()$**: sono primitive che agiscono su una estensione dei lock chiamata semaforo. Ogni semaforo $\text{sem}$ può assumere un qualunque valore positivo o nullo. Se il semaforo $\text{sem}$ vale 0, la primitiva $\text{P}(\text{sem})$ blocca il processo finché $\text{sem}$ non diventa positivo, altrimenti decrementa di 1 il suo valore. La primitiva $\text{V}(\text{sem})$ incrementa il valore di $\text{sem}$ e, nel caso in cui vi siano processi bloccati sul semaforo, sblocca uno di essi. Tali primitive consentono di risolvere in modo semplice ed elegante problemi del tipo “produttore/consumatore” con più produttori e/o più consumatori. In tale caso, infatti, si usano due appositi semafori che indicano, rispettivamente, quanti sono i buffer pieni e quanti sono i buffer vuoti. Come è ovvio, un produttore deve essere bloccato se non vi sono più buffer vuoti; in modo analogo, un consumatore deve essere bloccato se non vi sono più buffer pieni.
• **send() e receive()**: consentono a processi diversi di scambiarsi informazioni, garantendo allo stesso tempo la protezione degli spazi degli indirizzi di ogni processo. La risorsa consumabile è il *messaggio*, ossia una sequenza di caratteri spesso di lunghezza fissa, per motivi di efficienza. Il contenuto di un messaggio è arbitrario ed è stabilito dal processo mittente.

La primitiva `send(dest,m)` consente ad un processo mittente di inviare al processo destinatario `dest` un messaggio `m`. La primitiva `receive(mitt,m)` consente ad un processo di ricevere un messaggio. I parametri `mitt` e `m` specificano, rispettivamente, il nome del processo mittente ed il testo del messaggio. Tale coppia di primitive è realizzata associando ad ogni processo una coda di messaggi pendenti, ossia trasmessi e non ancora ricevuti.

Come nel caso della `unlock()` o della `V()`, la `send()` non può bloccare il processo che la esegue. Viceversa, se un processo `P` esegue una `receive()` e il NUCLEO verifica che non vi sono messaggi pendenti destinati a `P`, esso blocca `P` fino a quando un altro processo non esegue una `send()` destinata al processo `P`.

Le primitive `send()` e `receive()` sono utilizzate in tutti i contesti in cui la lunghezza del messaggio è limitata a qualche Kbyte. In effetti, i sistemi operativi che offrono tali primitive di sincronizzazione fanno uso di buffer interni per contenere i messaggi destinati ad un processo e, per motivi di efficienza, tali buffer sono di lunghezza fissa e di dimensioni alquanto limitate.

### 7.4 Come realizzare un programma concorrente

Come si scrive un programma concorrente in Unix? Vi sono diversi approcci possibili, ognuno dei quali offre particolari vantaggi e svantaggi.

Alcuni linguaggi di programmazione orientati ad oggetto come Java includono una classe standard di tipo thread i cui metodi realizzano tutte le classiche funzioni usate dai programmi concorrenti. Altri linguaggi orientati ad oggetto come il C++ non hanno una classe standard di tipo thread, anche se risulta abbastanza agevole realizzare una apposita classe che emula la classe thread di Java.

I linguaggi procedurali, infine, non hanno il concetto di classe. In questo caso, è necessario fare uso di apposite librerie di programmi che offrono tutte le funzioni necessarie per la programmazione concorrente.

La libreria Pthread per il linguaggio C, ad esempio, consente di creare proces-
### Tabella 7.1: Alcune funzioni della libreria Pthread

<table>
<thead>
<tr>
<th>funzione</th>
<th>descrizione</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create()</td>
<td>crea un nuovo thread</td>
</tr>
<tr>
<td>pthread_exit()</td>
<td>termina un thread</td>
</tr>
<tr>
<td>pthread_join()</td>
<td>aspetta la terminazione di un thread</td>
</tr>
<tr>
<td>pthread_mutex_lock()</td>
<td>richiede un lock</td>
</tr>
<tr>
<td>pthread_mutex_unlock()</td>
<td>rilascia un lock</td>
</tr>
<tr>
<td>pthread_cond_wait()</td>
<td>sospende il thread finché una variabile di tipo “condizione” non diventa vera</td>
</tr>
</tbody>
</table>

Si chiamati *thread* che condividono le variabili globali del processo padre. La tabella 7.4 elenca le principali funzioni offerte da tale libreria.

Poiché nostro obiettivo è lo studio dei sistemi operativi, ed in particolare dei sistemi operativi di tipo Unix, ci concentreremo sulle varie API, chiamate di sistema ed altri costrutti di programmazione che Unix offre al fine di realizzare programmi concorrenti. Inoltre, poiché Unix è codificato in buona parte nel linguaggio C, descriveremo il supporto Unix per la programmazione concorrente facendo riferimento al linguaggio C, anche se tale supporto è stato esteso ad altri linguaggi di programmazione quali Fortran, Cobol, ecc.

## 7.5 Supporto Unix

Il supporto Unix alla programmazione concorrente in C si esplica nelle seguenti funzionalità:

- **creazione/eliminazione di processi**: oltre alle API POSIX `fork()` ed `exit()`, Linux offre la chiamata di sistema `clone()` (vedi paragrafo 7.5.1) che non è inclusa nello standard POSIX.

- **segnali**: Unix prevede una serie di API POSIX per la gestione dei segnali (vedi paragrafo 7.5.2).

- **System V InterProcess Communication (IPC)**: le API POSIX incluse nel pacchetto IPC consentono di operare su semafori, di inviare o ricevere messaggi, e di condividere aree di memoria tra processi. Purtroppo, tali API sono piuttosto macchinose da utilizzare. Le più usate sono quelle di tipo `shmget()`, `shmat()`, e `shdet()` per gestire regioni di memoria condivise tra processi (vedi programmi *shared_memory*, esercitazione 5).
7.5. SUPPORTO UNIX

- **POSIX Message Queues**: un gruppo di API POSIX che consentono ai processi di scambiarsi messaggi con una interfaccia più semplice e più flessibile di quella usata dalle API IPC.

- **pipe**: sono un tipo di file speciale usato per lo scambio di dati tra processi. Il termine pipe sta ad indicare che si tratta di una semplice interazione del tipo produttore/consumatore tra il processo che scrive nella pipe e quello che legge dalla pipe. La sincronizzazione è a carico del sistema operativo: se la pipe è piena il produttore viene bloccato; in modo analogo, se la pipe è vuota, il consumatore viene bloccato (vedi programmi programmi `pipe.c` e `fifo.c`, esercitazione 4).

- **socket**: consentono la comunicazione bidirezionale tra programmi in esecuzione sullo stesso calcolatore oppure su calcolatori diversi. Il programma applicativo X Window utilizza socket per consentire lo scambio di dati tra i vari client ed il server (vedi paragrafo 5.4.1).

7.5.1 Creazione/eliminazione di processi

Descriviamo brevemente le API e chiamate di sistema che consentono di creare ed eliminare processi.

**Creazione di processi**

La chiamata di sistema

\[
\text{pid} = \text{fork}();
\]

è utilizzata per creare un processo figlio che è una copia conforme del processo padre. Come evidenziato dalla assenza di parametri, il processo figlio ottiene una copia di tutte le risorse possedute dal padre, e quindi una copia del programma da eseguire. Vi è un’unica differenza che può essere sfruttata dai programmatori per distinguere il figlio dal padre: il valore `pid` dell’identificatore di processo restituito dalla `fork()`. Esso vale 0 quando il codice è eseguito dal processo figlio, mentre ha un valore positivo quando è eseguito dal processo padre.

Lo schema di programma illustrato appresso mostra come il programmatore può sfruttare le caratteristiche della chiamata di sistema `fork()` per fare eseguire al processo figlio azioni diverse dalle azioni eseguite dal processo padre (vedi anche programma `fork.c` della esercitazione 4):
Si osservi che non soltanto le variabili locali ma anche quelle globali si riferiscono ad un singolo processo. Ne consegue che processi distinti non possono scambiarsi dati tramite variabili globali.

Può essere usata a posto della fork() la chiamata di sistema clone(): tale chiamata non è inclusa nello standard POSIX, per cui non è garantito che programmi che facciano uso della clone() possano essere ricompilati correttamente su altri sistemi Unix. In compenso, la clone() include tra i suoi parametri un vettore di flag che consentono di condividere diverse risorse tra il processo padre ed il processo figlio. In particolare, l’uso del flag CLONE_VM consente di condividere lo spazio degli indirizzi tra i due processi (vedi anche programmi crea_clone.c e crea_clone_malloc.c della esercitazione 4).

Nel seguente schema di programma:

```c
int clone_func(void *unused)
{
    .......
    return 0;
}

int main()
{
    pid_t pid;
    int stackcl[1024];
```
il processo padre esegue la funzione `main()`. Tale funzione usa la chiamata di sistema `clone()` per creare un processo figlio di tipo clone. Il flag `CLONE_VM` richiede la condivisione degli spazi degli indirizzi tra i due processi. Nell’esempio illustrato nel programma `2cloni.c` incluso nella esercitazione 4, questo significa che la variabile globale del processo padre `global_var` sarà condivisa dal clone.

Il primo parametro della `clone()` indica il nome della funzione che il processo figlio dovrà eseguire, il secondo parametro indica l’area di memoria usata dal clone come stack. Ogni clone deve avere il proprio stack. Inoltre, è necessario specificare come parametro della `clone()` l’indirizzo successivo all’ultimo byte dello stack e non quello del primo byte. Nell’esempio, `&stackcl[1024]` e non `&stackcl[0]`.

Linux realizza la libreria Pthread citata in precedenza facendo uso della chiamata di sistema `clone()`. La condivisione delle variabili globali da parte dei thread è assicurata grazie all’uso del flag `CLONE_VM`.

### Eliminazione di processi

La chiamata di sistema `_exit()` causa l’eliminazione del processo che la esegue. Tale chiamata di sistema dovrebbe essere usata dai programmatori C soltanto quando viene richiesta una terminazione prematura del programma mentre si sta eseguendo una qualche funzione diversa dalla funzione `main()`. La funzione `main()` deve invece terminare con una `return 0;` quando il compilatore C incontra la `return` che si riferisce alla funzione `main()` (oppure la parentesi graffa destra che chiude il codice della funzione), esso inserisce automaticamente una chiamata di sistema di tipo `_exit()` per fare terminare l’esecuzione del programma.

Nell’eseguire tale chiamata di sistema, il NUCLEO invia al processo padre un segnale di tipo `SIGCHLD` per segnalare che un processo figlio è terminato. Questo segnale viene raccolto dal processo padre (lo shell che ha lanciato il processo figlio) che provvede ad eliminare il figlio. In generale, Unix consente ad un processo di eliminare un altro processo inviando ad esso un apposito segnale (vedi paragrafo 7.5.2).

---

1 L’istruzione `push` decrementa prima il valore del registro puntatore allo stack (`%esp`) e scrive quindi all’indirizzo puntato dal registro.
7.5.2 Segnali

I **segnali** sono una tecnica di sincronizzazione alquanto rudimentale introdotta dal primo sistema Unix all’inizio degli anni ’70 e tuttora in uso. Un segnale non è altro che un identificatore che un processo può inviare ad un altro processo mediante la chiamata di sistema **kill()**.

Unix fa uso di un numero ristretto di segnali. I principali sono:

- **SIGKILL** per eliminare un processo.
- **SIGCHLD** per segnalare al processo padre che il figlio è terminato.
- **SIGINT** per segnalare una interruzione da tastiera.
- **SIGSTOP** e **SIGCONT** per sospendere e riprendere l’esecuzione di un processo.
- **SIGUSR1** e **SIGUSR2** disponibili per i programmi applicativi (vedi dopo).

Lo standard POSIX prevede inoltre alcune API per la gestione dei segnali:

- **kill()**: invia un segnale ad un processo o ad un gruppo di processi.
- **pause()**: blocca il processo finché esso non riceva un qualche segnale.
- **waitpid()**: blocca il processo finché un suo specifico processo figlio (oppure un qualsiasi processo figlio) non termini.
- **sigaction()**: associa una specifica funzione che verrà eseguita quando il processo riceverà un segnale del tipo prefissato. I programmi utenti possono fare uso della **sigaction()** per associare specifiche funzioni dipendenti dalla applicazione ai segnali **SIGUSR1** e **SIGUSR2**, vedi programma **2signals_incedec.c** illustrato nella esercitazione 5).
- **signal()**: simile a **sigaction()**, anche se il manuale suggerisce di usare **sigaction()**.
- **alarm()**: sospende il processo per un numero prefissato di secondi; scaduto tale intervallo di tempo, il NUCLEO sblocca il processo inviando ad esso un segnale **SIGALRM**\(^2\).

\(^2\)Il programma **my_sleep.c** introdotto nella esercitazione 5 illlustra come la API **sleep()** possa essere facilmente realizzata facendo uso della chiamata di sistema **alarm()**.
7.5. SUPPORTO UNIX

7.5.3 Una alternativa ai semafori delle API IPC

Notiamo da quanto detto finora che Unix non offre alcun supporto all’infuori delle API IPC per risolvere problemi di sincronizzazione del tipo “mutua esclusione”. In altre parole, a parte le farraginose API semct1(), semget() e semop() che devono essere usate per realizzare le semplici funzioni P() e V() descritte nel paragrafo 7.3, UNIX non offre alcun altro strumento per gestire tale problema di sincronizzazione.

Mostriamo in questo paragrafo come sia possibile realizzare le primitive di sincronizzazione lock() e unlock() assenti in Unix facendo uso di chiamate di sistema esistenti.

L’idea di base per realizzare lock() è quella di sfruttare le caratteristiche del file system Unix, ed in particolare delle chiamate di sistema open() con i relativi parametri. Se infatti un file è stato aperto da un processo con il parametro O_EXCL, nessun altro processo può aprire il file finché il file non viene chiuso dal processo che lo ha aperto in precedenza.

Possiamo quindi usare i nomi di file come lock: se il file esiste, il lock corrispondente è impegnato, altrimenti risulta libero.

La funzione lock(const char *p) illustrata nel programma lock_unlock.c della esercitazione 5 opera su un singolo parametro p che è un puntatore al nome del file, ossia al nome del lock che si intende utilizzare; tale nome può essere sia un pathname assoluto che relativo.

Come si può osservare dal codice, la funzione lock() tenta di aprire il file con modalità esclusiva O_EXCL: se un file con lo stesso nome già esiste, lock() rilascia temporaneamente la CPU eseguendo la chiamata di sistema sched_yield() che pone il processo in nello stato di pronto. Successivamente, quando lo stesso processo andrà in esecuzione, verrà nuovamente eseguita la open() finché l’apertura del file riesce (ciclo while).

Quando il file è stato creato e quindi aperto, lock() termina restituendo il file descriptor fd del file corrispondente al lock ottenuto.

La funzione unlock(const char *p, int fd) illustrata nello stesso programma opera su due parametri: p è un puntatore al nome del lock ottenuto in precedenza mentre fd è il file descriptor associato al file aperto.

Come si può osservare dal codice, la funzione unlock() inizia col chiudere il file corrispondente al lock. Se tale file non esiste (uso improprio dei lock), segnala una condizione di errore tramite il codice di ritorno. Altrimenti, la funzione unlock()
92  CAPITOLO 7. PROGRAMMAZIONE CONCORRENTE

provvede ad eliminare il file corrispondente al lock invocando la chiamata di sistema `unlink()`. Poiché il file è stato aperto da un solo processo, il contatore d’uso del file verrà posto a 0 in seguito alla `unlink()` ed il NUCLEO provvederà quindi a cancellare il file dal file system.

Si osservi il ruolo del primo parametro `p`: esso punta al nome del lock ed è necessario per potere invocare la chiamata di sistema `unlink()`. Non esiste infatti in Unix una funzione che consenta di derivare dal file descriptor il corrispondente nome del file.

Molte applicazioni per sistemi Unix usano l’approccio appena illustrato per realizzare primitive di sincronizzazione di tipo `lock()/unlock()`: l’interfaccia grafica X Window, ad esempio, fa uso di un file di lock chiamato: `.X0-lock`. Quando il server X Window è stato avviato da un utente, viene creato questo file nella directory `/tmp`. Il file viene cancellato solo quando l’utente esce dall’applicazione X Window.

7.6  Stallo tra processi

Non si può parlare di programmazione concorrente senza accennare agli errori di sincronizzazione, ossia ad errori di programmazione causati dall’uso improprio delle primitive di sincronizzazione.

L’errore di interazione più vistoso e più noto prende il nome di stallo o deadlock: durante l’avanzamento del gruppo di processi tra loro interagenti, uno o più processi vengono posti in stato di attesa e vi rimangono per un tempo indeterminato.

La figura 7.1 illustra un classico caso di programma concorrente che può dare luogo, in determinate circostanze, ad uno stallo tra due processi.

Tale errore è causato da un uso improprio delle primitive di sincronizzazione `lock()` e `unlock()` da parte dei processi: i due processi `P` e `P’` fanno uso entrambi della stessa coppia di risorse che chiameremo `R1` ed `R2`. Per garantire l’accesso seriale ad ognuna di esse i processi fanno uso di due lock chiamati `lock_R1` e `lock_R2` e delle relative primitive di sincronizzazione `lock()` e `unlock()` descritte in precedenza.

Il processo `P` esegue, ad esempio, una `lock(lock_R1)` prima di accedere ad `R1` ed una `unlock(lock_R1, fd)` per rilasciare `R1`.

Il problema consiste nel fatto che `P` richiede prima il semaforo `lock_R1` e poi
7.7. PROGRAMMAZIONE IN TEMPO REALE

Figura 7.1: Esempio di programma che può causare stallo.

quello lock_R2, mentre il processo \( P' \) richiede prima il semaforo lock_R2 e poi quello lock_R1. Può quindi succedere che, in seguito allo scheduling deciso dal NUCLEO per \( P \) e \( P' \), \( P \) ottenga il semaforo lock_R1 e subito dopo \( P' \) ottenga il semaforo lock_R2.

A questo punto si determina una condizione di stallo poiché né \( P \) né \( P' \) sono in grado di continuare la loro esecuzione: ognuno aspetta un semaforo posseduto dall’altro.

Come si evitano errori di interazione di tipo stallo in un programma concorrente? La risposta sta nell’esempio appena illustrato: bisogna obbligare i processi a richiedere semafori in un ordine prefissato, ad esempio in un ordine crescente. È compito del programmatore, e non del sistema operativo, scrivere programmi concorrenti che non contengano errori di interazione.

7.7 Programmazione in tempo reale

Concludiamo questo capitolo sulla programmazione concorrente dando alcuni cenni su una variante di programma concorrente che è molto utilizzata nell’ambito del controllo di processi.

Un programma in tempo reale è un programma concorrente che deve soddisfare dei vincoli relativi ai tempi d’esecuzione di almeno uno dei sottoprogrammi che lo compongono. Programmi di tale tipo sono anche chiamati programmi dipendenti dal tempo.

Un tipico esempio di programma in tempo reale è quello per il controllo di un processo industriale: se il sistema riceve dall’esterno segnali corrispondenti a una situazione che richiede un intervento tempestivo, esso non deve soltanto
ricongoscere tale situazione ma anche produrre le informazioni richieste entro e non oltre un intervallo prefissato di tempo.

Un altro esempio è quello di un programma per le previsioni metereologiche: in tale caso, il tempo richiesto per eseguire la previsione deve essere inferiore all’inverso della frequenza di rilevazione dei dati metereologici. Anche i sistemi operativi sono, sia pure in misura diversa, dei programmi in tempo reale.

I due estremi della gamma possono essere rappresentati, da un lato da un semplice sistema interattivo senza particolari vincoli sui tempi di risposta all’utente e, dall’altro, da un sistema dedicato per il controllo di una centrale telefonica elettronica.

Nel primo caso, l’unica parte dipendente dal tempo sono i sottoprogrammi che gestiscono le diverse interruzioni. Nel secondo caso, invece, quasi tutti i sottoprogrammi sono dipendenti dal tempo in quanto ogni ritardo nel trattare una coppia di linee tra cui è in svolgimento una comunicazione si traduce in un malfunzionamento del sistema.

In generale, i sistemi operativi “general purpose” non si prestano ad eseguire correttamente programmi in tempo reale e si preferisce fare uso di sistemi operativi semplificati di tipo hard real time in grado di assicurare tempi di risposta brevi ai processi che lo richiedono.
Capitolo 8

SPAZIO DEGLI INDIRIZZI DI UN PROCESSO

8.1 Introduzione

Ancora una volta, l’approccio “top down” ci forza a considerare la problematica della gestione della memoria in modo diverso da quello tradizionale. Anziché cominciare dal NUCLEO e descrivere le varie tecniche da esso utilizzate per gestire la memoria RAM, partiamo dai processi ed esaminiamo i motivi per cui i processi richiedono o rilasciano memoria; in particolare, studiamo:

- gli eventi più significativi che inducono un processo a richiedere memoria;
- i vari modi, diretti o indiretti, tramite i quali un processo effettua richieste di memoria al NUCLEO.

Rimandiamo invece al prossimo capitolo la discussione di come il NUCLEO assegna effettivamente RAM ai processi.

8.2 Spazio degli indirizzi di un processo

Ci riferiamo in questo contesto a file eseguibili contenenti programmi codificati in linguaggio macchina e pronti ad essere eseguiti dalla CPU. Chiamiamo indirizzo logico un gruppo di bit che serve a identificare:
96  
CAPITOLO 8. SPAZIO DEGLI INDIRIZZI DI UN PROCESSO

- l’indirizzo della prossima istruzione da eseguire;

oppure:

- l’indirizzo di un operando contenuto in RAM (solo nel caso in cui l’istruzione debba accedere ad operandi contenuti nella RAM).

Per semplicità, supponiamo che l’indirizzo logico consista in un gruppo di 32 bit\(^1\). Negli esempi successivi, rappresenteremo quindi gli indirizzi logici come numeri inclusi nell’intervallo compreso tra 0x00000000 e 0xffffffff.

Chiamiamo invece indirizzo fisico un gruppo di bit usati dal bus per indirizzare una cella di RAM. I microprocessori della famiglia Intel 80x86 che usiamo come esempio in queste dispense fanno uso di indirizzi fisici da 32 bit\(^2\).

Come vedremo nel capitolo successivo, un compito fondamentale del NUCLEO consiste nel definire un mapping tra indirizzi logici ed indirizzi fisici, per cui durante l’esecuzione di una istruzione, ogni indirizzo logico viene tradotto in un opportuno indirizzo fisico.

Il programmatore può in qualche modo determinare gli indirizzi logici usati dal suo programma ma ignora quali saranno gli indirizzi fisici assegnati al programma durante l’esecuzione. In modo analogo, un compilatore, un assemblatore o un linker possono impostare gli indirizzi logici dei programmi ma non quelli fisici.

Iniziamo col descrivere come il linker (vedi paragrafo 3.3) assegna indirizzi logici al programma da eseguire. Procediamo per gradi e supponiamo dapprima che il file eseguibile sia stato linkato in modo statico, ossia che non vengano utilizzate librerie dinamiche.

In questo caso, il linker statico assegna al processo che dovrà eseguire il programma un gruppo di regioni di memoria, ossia intervalli di indirizzi logici, caratterizzate da un indirizzo iniziale e da una lunghezza. Ad un file eseguibile corrispondono quindi diverse regioni di memoria. Per motivi di flessibilità, le regioni di memoria assegnate ad un processo possono essere tra loro non contigue.

\(^1\)In realtà, diversi microprocessori tra cui quelli Intel 80x86, fanno uso di indirizzi logici più complessi composti da due componenti: un segmento da 16 bit ed un offset da 32 bit, l’indirizzo risultante chiamato “indirizzo lineare” è comunque un indirizzo da 32 bit.

\(^2\)I microprocessori che includono registri a 64 bit usano indirizzi logici da 64 bit ed indirizzi fisici di più di 32 bit, ad esempio 48 bit per l’architettura x86_64.
8.2. SPAZIO DEGLI INDIRIZZI DI UN PROCESSO

Figura 8.1: Spazio degli indirizzi di un processo.

L’insieme di indirizzi logici racchiuso nelle regioni di memoria di un processo prende il nome di spazio degli indirizzi del processo: esso include tutti e soli gli indirizzi logici che un processo è autorizzato ad usare durante la sua esecuzione.

Ogni indirizzamento effettuato dal processo al di fuori dal suo spazio degli indirizzi viene considerato dal NUCLEO come un indirizzamento non valido dovuto ad un errore di programmazione.\(^3\)

La figura 8.1 illustra un semplice esempio di spazio degli indirizzi di un processo.

Come si osserva dalla figura, in base alle indicazioni fornite dal linker sono state assegnate al programma 4 regioni di memoria distinte tra loro non contigue:

- **regione codice**: contiene le istruzioni del programma, è lunga 64 KB ed inizia all’indirizzo logico 0x00800000;
- **regione dati inizializzati**: contiene costanti, è lunga 64 KB ed inizia all’indirizzo logico 0x00c00000
- **regione dati non inizializzati**: contiene lo heap (memoria dinamica ottenibile tramite la API *malloc*(), è lunga 128 KB ed inizia all’indirizzo logico 0x00f00000

\(^3\)Una eccezione a tale regola vale per la regione di memoria che contiene lo stack: in tale caso, è possibile che, in seguito ad una serie di *push*, il processo effetti un indirizzamento immediatamente prima dell’inizio della regione (lo stack cresce tradizionalmente per indirizzi decrescenti). In tale caso, il NUCLEO espande la regione stack e ri-esegue l’istruzione *push*. 

**CAPITOLO 8. SPAZIO DEGLI INDIRIZZI DI UN PROCESSO**

- *regione stack*: contiene lo stack, inizia al massimo indirizzo logico possibile, ossia 0xffffffff e si espande all’ingiù.⁴

### 8.3 Assegnazione di regioni di memoria

Il *loader* è il componente del sistema operativo che assegna regioni di memoria ad un nuovo processo. Tipicamente, il loader assegna tre tipi di regioni di memoria diversi al processo:

- un gruppo di regioni di memoria corrispondente alle diverse parti del file eseguibile che è stato linkato staticamente (vedi sopra);
- un gruppo di regioni di memoria usato dalla libreria ld-linux.so.2: tale libreria contiene il codice necessario per caricare tutte le altre librerie dinamiche specificate nel file eseguibile nonché il codice del programma *dl_resolve* che effettua il linkaggio di simboli non risolti dal liker statico durante l’esecuzione del programma (vedi paragrafo 3.3.2)
- un gruppo di regioni di memoria per ognuna delle librerie dinamiche specificate durante la fase di linkaggio (opzione *-lxxx* del gcc).

Anche se il processo utilizza una sola funzione di una libreria che ne contiene diverse migliaia, il loader assegna al processo regioni di memoria (regione codice e regioni dati) in grado di contenere l’intera libreria. Questo non costituisce uno spreco di memoria in quanto la risorsa assegnata al processo consiste in intervalli di indirizzi logici. L’assegnazione di RAM avviene durante l’esecuzione del processo mediante la tecnica del “demand paging” che verrà descritta nel paragrafo 10.5.

Per esaminare le regioni di memoria assegnate ad un processo, suggeriamo il seguente approccio:

1. scrivere un programma che si blocca durante l’esecuzione, ad esempio in seguito ad una *getchar()*;

2. compilare e linkare il programma e mandarlo in esecuzione;

⁴In Linux, l’ultimo GB di indirizzi lineari è riservato al NUCLEO, per cui lo stack di un processo User Mode inizia all’indirizzo 0xbfffffff.
3. quando il programma si blocca, usare un’altro programma terminale (xterm) e lanciare il comando: `ps ax`;

4. determinare il pid del programma di prova attualmente stoppato;

5. lanciare il comando: `cat /proc/pid/maps` che visualizza tutte le regioni di memoria attualmente possedute dal processo pid.

### 8.4 Modifiche dello spazio degli indirizzi

Vediamo ora come lo spazio degli indirizzi iniziale di un processo possa cambiare durante l’esecuzione del processo: il processo può acquisire nuove regioni di memoria, allargare o restringere la dimensione di regioni di memoria esistenti, o rilasciare regioni di memoria.

Ancora una volta, ci interessiamo alle regioni di memoria e quindi agli indirizzi logici posseduti da un processo e non al come il NUCLEO assegna indirizzi fisici ai vari indirizzi logici.

La regione “codice” e quella “dati inizializzati” rimangono solitamente immutate durante l’esecuzione del programma. Vice versa, la regione “dati non inizializzati” può crescere in seguito a richieste di memoria dinamica da parte del processo. In modo analogo, la regione “stack” può crescere durante l’esecuzione del programma (ad esempio, per causa dell’annidamento di chiamate di funzioni).

Esiste tuttavia un caso in cui il processo rilascia tutte le regioni di memoria da esso possedute, incluse la regione “codice” e quella “dati inizializzati”, ed ottiene un nuovo gruppo di regioni del tutto diverse da quelle ottenute in precedenza. Questo avviene quando il processo effettua una chiamata di sistema per caricare dinamicamente un file eseguibile.

Nei sistemi Unix, esistono diverse chiamate di sistema del tipo `exec-like` che realizzano ciò: `execl()`, `execlp()`, `execle()`, `execv()`, `fexecvp()`, `execve()`, `fexecve()`. Il programma fork-execve() della esercitazione 4 illustra un esempio d’uso di tale chiamata di sistema.

Dal punto di vista della programmazione, la `execve()` corrisponde ad un salto senza ritorno, per cui la `printf()` inclusa nel programma viene eseguita solo nel caso in cui la chiamata di sistema non sia andata a buon fine.

Abbiamo visto in precedenza che quando il file eseguibile è stato linkato a librerie
dinamiche, il loader assegna regioni di memoria non solo al programma da eseguire ma anche ad un modulo del sistema operativo chiamato linker dinamico.

Abbiamo anche descritto sommariamente nel paragrafo 3.3.2 come il linker dinamico svolga ulteriori linkaggi del programma in base alle chiamate alle funzioni di libreria effettuate durante l’esecuzione. In tale contesto, il linker dinamico assume che le regioni di memoria usate dalla libreria siano già state allocate in precedenza dal loader.

Nei sistemi operativi POSIX-compliant, è anche possibile richiedere durante l’esecuzione del processo anziché prima dell’esecuzione la creazione di ulteriori regioni di memoria per contenere una nuova libreria dinamica. Ciò si realizza mediante la chiamata di sistema dlopen(): ogni volta che il programma invoca tale chiamata di sistema, entra in funzione un programma contenuto in un’apposita libreria chiamata libdl.so.2 per assegnare nuove regioni di memoria per la libreria specificata come parametro della dlopen().

Si noti come, a differenza della execve(), la dlopen() corrisponde ad una chiamata di funzione con ritorno al programma chiamante.

Il programma dlopen.c della esercitazione 4 illustra l’uso della chiamata di sistema dlopen().

Per essere linkato correttamente, bisogna specificare tra i moduli oggetto da linkare anche quelli contenuti nella libreria libdl.so.2 citata in precedenza. Ciò si ottiene facendo uso dell’opzione -l seguita dalla sigla dl della libreria dinamica, per cui il comando gcc deve essere del tipo:

```
gcc -o prova -Wall prova.c -ldl
```

8.5 **Mapping di file in memoria**

Il concetto di regione di memoria rischia di rimanere alquanto vago se non si specifica in che modo gli indirizzi logici di una regione sono collegati alle varie parti del file eseguibile. Questo è precisamente il ruolo del cosiddetto *mapping di file in memoria*: associare ad una regione di memoria un file (o parte di esso) residente su disco.

In pratica, la stessa chiamata di sistema mmap() utilizzata per creare una regione di memoria prevede tra i suoi parametri un parametro di tipo “file descriptor”
che identifica un file aperto in precedenza ed un altro paio di parametri che specificano la porzione del file da mappare.

A questo punto, la distinzione tra spazio degli indirizzi e RAM assegnata al processo dovrebbe risultare chiara: a mano a mano che il processo esegue istruzioni, esso fa uso di nuovi indirizzi logici ed è compito del NUCLEO provvedere sia ad assegnare nuova RAM al processo, sia a leggere da disco nell’area di RAM ottenuta i dati associati agli indirizzi logici. Tutto ciò avviene in modo trasparente rispetto al programma: come detto in precedenza, il programmatore non ha alcun controllo sugli indirizzi fisici usati dal suo programma ed è cura del NUCLEO assegnare (o rilasciare) aree di RAM al processo che esegue il programma.

Come vedremo nel prossimo capitolo, la strategia seguita dal NUCLEO consiste nell’allocare RAM al processo il più tardi possibile, ossia quando il processo effettua indirizziamenti in una zona di regione di memoria non ancora caricata in memoria. Questo approccio è di solito efficace in quanto molti processi eseguono soltanto un sottoinsieme delle istruzioni contenute nel programma.

8.5.1 Tipi di mapping

Tra i suoi vari parametri, la chiamata di sistema `mmap()` ne prevede uno che specifica il tipo di mapping. Si tratta di un flag che può assumere diversi valori. Ci limitiamo a descrivere tre chiamati `MAP_SHARED`, `MAP_PRIVATE` e `MAP_ANONYMOUS`.

- **MAP_SHARED**: la regione di memoria ha una immagine corrispondente su disco e il NUCLEO ha l’obbligo di aggiornare l’immagine su disco ogni volta che un processo modifica il contenuto della regione di memoria.

- **MAP_PRIVATE**: simile a MAP_SHARED ma il NUCLEO non aggiorna l’immagine su disco.

- **MAP_ANONYMOUS**: la regione di memoria non ha una immagine corrispondente su disco ed il NUCLEO non deve intervenire quando un processo modifica il contenuto della regione di memoria. Come illustrato nella Figura 8.1, le regioni di memoria che non hanno una immagine su disco, ossia un file di riferimento da cui estrarre le informazioni richieste, sono la regione stack e quella usata dallo heap. In entrambi i casi, le regioni nascono vuote ed è il processo a dovere scrivere dati in esse prima di poterli rileggere: lo stack di un processo è inizialmente vuoto e si riempie a mano a mano che il processo invoca funzioni le quali a loro volta invocano altre funzioni (i compilatori C collocano tutte le variabili locali ad una funzione in cima allo
stack). In modo analogo, l’area di memoria dinamica ottenuta tramite la API `malloc()` non è inizializzata.

La distinzione tra i tre tipi di mapping è importante dal punto di vista del NUCLEO: quando un processo effettua una scrittura in una delle sue regioni di memoria, le azioni svolte dal NUCLEO sono diverse a seconda del tipo.

### 8.5.2 Mapping di un file di dati

Esiste un altra applicazione del mapping di file in memoria non illustrata nella Figura 8.1: il mapping di file di dati. Tale approccio consente di accedere ai dati di un file di dati come se fossero mappati dentro ad un array, anziché essere contenuti all’interno del file. Nel caso di indirizzamenti casuali del file, non è quindi necessario fare uso della chiamata di sistema `lseek()` ma è sufficiente indirizzare la voce richiesta dell’array.

Il programma `memmap.c` contenuto nella esercitazione 4 illustra un semplice esempio di mapping di file di dati.
Capitolo 9

STRUTTURA INTERNA DEL NUCLEO

9.1 Introduzione

Dagli anni ’60, quando apparvero i primi sistemi operativi multiprogrammati, ad oggi, le metodologie di progetto si sono affinate fino a giungere ad un buon livello di standardizzazione per quanto riguarda la realizzazione di NUCLEI per architetture a memoria centralizzata condivisa. Sono invece tuttora in corso ricerche e approfondimenti per quanto riguarda la progettazione di sistemi operativi per architetture multiprocessore (sistemi con più CPU) nonché per architetture distribuite (reti di calcolatori, basi di dati distribuite, ecc.). In questo capitolo, si esaminano brevemente alcune delle problematiche più significative attinenti al progetto di NUCLEI di sistemi operativi multiprogrammati mentre, nel capitolo successivo, si descrivono le funzioni svolte dai principali moduli inclusi nel NUCLEO.

9.2 NUCLEO e processi

L’uso della combinazione NUCLEO/processi discusso nel capitolo 6 risulta determinante per realizzare in modo semplice ed elegante sistemi operativi di tipo multitasking.

Come detto in precedenza, il NUCLEO deve essere considerato come uno strato intermedio di software tra i programmi più complessi del sistema operativo.
CAPITOLO 9. STRUTTURA INTERNA DEL NUCLEO

(interprete di comandi, compilatori, ecc.) e l’hardware dell’elaboratore.

In base a questa impostazione modulare, i rimanenti programmi del sistema operativo vedono la combinazione hardware + NUCLEO come una macchina virtuale più facile da programmare della macchina reale. Per questo motivo, il NUCLEO è talvolta considerato come una estensione diretta dell’hardware.

Va tuttavia messo in evidenza che esistono alcuni compiti fondamentali, quelli che garantiscono appunto la corretta gestione dei processi, che devono necessariamente essere realizzati dal NUCLEO in quanto coordinatore dell’avanzamento dei processi. In particolare, citiamo:

- la creazione ed eliminazione di processi;
- la commutazione di processi (in inglese, task switching). Tale commutazione consiste nel porre in un opportuno stato di attesa il processo in esecuzione e porre quindi in esecuzione il processo selezionato dalla funzione di scheduling;
- la gestione di tipo time sharing del tempo di CPU con i relativi programmi che realizzano l’algoritmo di scheduling;
- la realizzazione di opportuni schemi di interazione tra processi (vedi paragrafo 7.5);

Oltre a creare processi, coordinare l’avanzamento ed eliminarli quando necessario, il NUCLEO deve svolgere numerose altre attività quali:

- la inizializzazione del sistema operativo con relativa impostazione delle strutture di dati necessarie e la creazione dei primi processi di sistema;
- la gestione dei segnali d’interruzione provenienti sia dalla CPU che da dispositivi esterni;
- la chiusura ordinata del sistema operativo che precede lo spegnimento dell’elaboratore con il rilascio di tutte le risorse possedute dai processi e la loro eliminazione.

Accenneremo nei prossimi paragrafi al modo in cui NUCLEO svolge i diversi compiti ad esso assegnati. Per ora, limitiamoci a studiare che cosa rende il programma NUCLEO diverso dagli altri programmi finora considerati.
9.3 Che cosa è il NUCLEO

Come i normali programmi, il NUCLEO deve essere opportunamente compilato e linkato dando luogo ad un file eseguibile. Tale file eseguibile non può tuttavia essere caricato utilizzando un loader poiché esso deve essere caricato subito dopo l’accensione del sistema quando la RAM non è stata ancora inizializzata.

La tecnica usata per caricare il NUCLEO in RAM viene chiamata bootstraping: terminato il Power On Self Test, ossia il riconoscimento dei dispositivi hardware collegati al sistema, la CPU esegue un programma registrato in una memoria non volatile ad un indirizzo prefissato. Tale programma trasferisce in RAM i primi 512 byte del dispositivo di boot (disco rigido, CD-ROM, ecc.) e passa ad eseguire il codice contenuto in tali byte (bootloader primario). Tale programma contiene istruzioni le quali, a loro volta, provvedono a leggere il bootloader secondario e a passare il controllo ad esso. Il bootloader secondario infine carica in RAM l’immagine del NUCLEO scelta dall’utente tramite una primitiva interfaccia a menu e passa il controllo alla funzione di inizializzazione.

Il NUCLEO inizializza le proprie strutture di dati e crea alcuni processi di servizio, tra cui i processi di tipo “login shell” in grado di accettare comandi dagli utenti. Al termine di tale fase di inizializzazione, il NUCLEO diventa operativo.

Quando il NUCLEO è operativo, esso è pronto sia a soddisfare richieste da parte dei processi in esecuzione, sia a registrare nelle proprie strutture di dati il verificarsi di eventi causati da dispositivi di I/O esterni alla CPU. In altre parole, il NUCLEO è in grado di rispondere in tempi brevi a richieste di servizio provenienti sia dalla CPU che da altri dispositivi hardware.

Per capire come ciò sia possibile, è necessario fare riferimento ai segnali di interruzione e al supporto hardware offerto dal calcolatore per realizzare il modello NUCLEO/processi.

Come illustrato in Figura 9.1, il NUCLEO è un programma di tipo “interrupt driven”, ossia un gruppo di programmi in qualche modo indipendenti che sono attivati soltanto quando si verificano interruzioni di uno specifico tipo. Ognuno di tali programmi indipendenti possiede un proprio “entry point”, ossia l’indirizzo della prima istruzione del programma.

Poiché i moderni calcolatori prevedono decine di interruzioni diverse, vi sono decine di “entry point” tramite i quali possono essere eseguiti i corrispondenti programmi del NUCLEO.

Quando l’unità di controllo della CPU rileva un segnale di interruzione, essa:
1. identifica il numero d’ordine \( i \) dell’interruzione;

2. passa in Kernel Mode ed usa lo stack Kernel per salvare il contenuto di alcuni registri del processo sospeso;

3. usa la tabella dei descrittori delle interruzioni ed il numero \( i \) per derivare l’indirizzo o “entry point” del programma del NUCLEO atto a trattare interruzioni di tipo \( i \);

4. imposta nel registro contatore programma l’indirizzo derivato al passo precedente.

Al termine delle quattro azioni svolte dall’unità di controllo, la CPU passa ad eseguire la prima istruzione del programma del modulo atto a trattare interruzioni di tipo \( i \). In altre parole, viene eseguito il programma del NUCLEO avente l’“entry point” corrispondente alla interruzione che si è appena verificata.

Per motivi di flessibilità, sia l’indirizzo di base della tabella dei descrittori delle interruzioni, sia i valori degli “entry point” possono essere impostati dal NUCLEO durante la fase di installazione, ossia prima di avere abilitato le interruzioni.

L’architettura Intel 80x86 include, ad esempio, un registro \texttt{idtr} che punta alla base della tabella dei descrittori delle interruzioni. Ogni descrittore lungo 8 byte contiene l’indirizzo logico dell’“entry point” nonché altri flag per la protezione (vedi Manuali Intel per ulteriori dettagli).
9.4 Ruolo dei segnali di interruzione

I segnali di interruzione sono generati sia dall’unità di controllo della CPU, sia da dispositivi esterni alla CPU\(^1\), per rappresentare eventi tra loro molto diversi quali:

- segnalazione da parte di un dispositivo esterno della fine di una operazione di ingresso/uscita;
- segnalazione da parte del chip interval timer: terminazione di un intervallo di tempo prefissato;
- segnalazione da parte dell’unità di controllo della CPU di una condizione anomala verificatasi nell’eseguire una istruzione: codice operativo non valido, indirizzo operando non valido, indirizzo istruzione non valido, ecc.;
- esecuzione di una apposita istruzione il cui effetto è quello di generare un segnale di interruzione (vedi paragrafo 9.6);
- segnalazione di un evento ad una o più CPU (solo nei sistemi multiprocessor).

9.5 Gestori delle interruzioni

I programmi del NUCLEO che devono trattare specifiche interruzioni prendono il nome di *gestori delle interruzioni* (in inglese, *interrupt handlers*).

In generale, ogni segnale di interruzione richiede un suo apposito gestore: il gestore delle interruzioni emesse periodicamente dal circuito hardware interval timer, ad esempio, non ha niente in comune con quello che gestisce le interruzioni provenienti da tastiera.

Come vedremo nel prossimo capitolo su specifici esempi, il verificarsi di un determinato segnale di interruzione può causare cambiamenti di stato tra i processi in vita nel sistema.

Un classico caso è quello delle operazioni di I/O “bloccanti”, ossia operazioni che devono essere eseguite prima che il processo possa continuare ad eseguire istruzioni. Se, ad esempio, il processo effettua una chiamata di sistema `read()`

\(^1\)La Intel denota come “exception” una segnale di interruzione prodotto dalla CPU e come “interrupt” un segnale prodotto da un dispositivo diverso dalla CPU.
per leggere dati da disco, esso dovrà essere bloccato dal NUCLEO fino a quando la lettura dei dati non sarà avvenuta poiché la semantica delle varie forme di `read()` presenti nei linguaggi di programmazione specifica che i dati letti sono immediatamente disponibili per altre elaborazioni da parte del programma.

In questo specifico caso, il segnale di interruzione emesso dal disco per segnalare l'avvenuta lettura dei dati è l'evento atteso dal processo bloccato, per cui tale segnale può essere considerato come una risorsa consumabile prodotta dal controllore del disco e consumata dal processo bloccato sull'operazione di `read()`.

In generale, però, non esiste sempre un processo bloccato per ogni segnale di interruzione che si verifica. Nel caso dei segnali di interruzione emessi periodicamente dall'interval timer, ad esempio, non vi è alcun processo bloccato in attesa di tale evento.

### 9.6 Chiamate di sistema

La criticità delle funzioni svolte fa sì che il passaggio di controllo ad un qualsiasi programma del NUCLEO non è realizzato come una normale chiamata di procedura, bensì tramite una tecnica speciale che consente non soltanto di passare il controllo ma anche di assicurare la protezione dei programmi e dei dati contenuti nel NUCLEO.

Per distinguerele dalle semplici chiamate di procedura, le chiamate a programmi del NUCLEO prendono il nome di *chiamate di sistema*.

In molte architetture, incluse quelle dei microprocessori più recenti, le chiamate di sistema sono realizzate tramite una apposita istruzione di tipo `int i` (il nome specifico dipende dal tipo di calcolatore utilizzato) che fa passare la CPU dallo stato User Mode allo stato Kernel Mode, generando una apposita eccezione chiamata *interruzione software* o interruzione di programma. Il parametro `i` della istruzione `int` identifica il tipo di interruzione che verrà generata dall'hardware.

In Linux viene usato il parametro 128 per generare apposite interruzioni che consentono a programmi User Mode di invocare un componente del NUCLEO chiamato *gestore delle chiamate di sistema*.

Come è ovvio, il programma deve anche comunicare a tale gestore quale tra le centinaia di chiamate di sistema deve essere eseguita e quali sono gli eventuali parametri sui quali la chiamata di sistema deve agire. In Linux, il passaggio di parametri avviene impostando nei registri della CPU `eax`, `ebx`, `ecx`, ecc. l'identi-
ficatore della chiamata di sistema e gli eventuali parametri di tale chiamata (vedi programma my_read.c nella esercitazione 5).

In questo modo è assicurata la protezione dei programmi del NUCLEO e delle relative strutture di dati poiché l’unico modo di accedere ad esso da parte di un programma in esecuzione è tramite l’esecuzione della suddetta istruzione int.

Quando si verifica una interruzione software, la CPU passa automaticamente ad eseguire le istruzioni del gestore delle chiamate di sistema, ossia dell’apposito programma del NUCLEO abilitato a trattare simili eventi. Il gestore verifica il tipo di chiamata, gli eventuali parametri associati alla chiamata e, nel caso in cui tutto sia regolare, passa il controllo al programma che gestisce la chiamata di sistema il cui numero è contenuto nel registro eax.

Quando il NUCLEO ha svolto le azioni richieste, esso esegue una apposita istruzione di tipo iret, il cui effetto è quello di ripristinare il contesto in stato User Mode del processo che aveva effettuato la chiamata di sistema e di saltare all’indirizzo salvatoo in precedenza sullo stack Kernel Mode del processo. Dopo avere eseguito la iret, la CPU passa quindi ad eseguire l’istruzione successiva alla int.

9.7 Descrittori di risorse

Per svolgere le sue attività, il NUCLEO fa uso di numerosi descrittori di risorse, ossia di strutture di dati di vario tipo quali liste, tabelle, vettori di tabelle, ecc.

Esempi classici di descrittori si riferiscono alle seguenti risorse:

- **processo**: ogni processo gestito dal NUCLEO possiede un proprio descrittore che deve essere aggiornato ogni qualvolta il processo cambi cambiato stato.

- **memoria RAM**: ogni blocco (page frame) di memoria di 4 KB possiede un descrittore di 32 B che specifica l’eventuale uso da parte di uno o più processi (nel caso di blocchi condivisi)

- **spazio degli indirizzi di un processo**: ogni processo è abilitato a fare uso di uno specifico insieme di indirizzi lineari, a prescindere dal fatto che le pagine di dati associate a tali indirizzi siano effettivamente presenti in RAM. Come abbiamo visto nel paragrafo 8.2, il NUCLEO fa uso di un apposito descrittore per ogni regione di memoria assegnata ad un processo.

- **memoria secondaria**: il NUCLEO mantiene per ogni file system montato alcuni descrittori che rappresentano il contenuto del disco; anche se tali
descrittori sono già presenti sul disco stesso, risulta molto più efficiente farne una copia in memoria per ridurre il numero di accessi al disco. Nei file system Unix, tali descrittori prendono il nome di “superblocco”, “descrittori di gruppi di blocco” e “inode”.

- **file aperti da un processo**: l’interazione tra un processo ed un file richiede l’uso di altri descrittori che contengono diverse informazioni quali il modo in cui è stato aperto il file, il byte del file attualmente scandito dal processo, ecc. Nei file system Unix, tali descrittori prendono il nome di “file object”.

A prescindere dal modo in cui sono realizzati, i descrittori di risorsa hanno alcune importanti proprietà in comune.

- **ogni descrittore è memorizzato in RAM in un’area di memoria riservata al NUCLEO**: per motivi di sicurezza, i processi non devono potere accedere direttamente ad un descrittore di risorsa. Possono invece accedere in lettura ed, in pochi casi, in scrittura ad alcune variabili del NUCLEO tramite pseudo-filesystem quali il procfs (cartella /proc o il sysfs (cartella /sys).

- **ogni descrittore utilizzato dal NUCLEO deve essere opportunamente inizializzato prima di poterlo utilizzare**: In generale, durante l’inizializzazione del sistema operativo, vengono riservate le aree di memoria destinate a contenere i vari descrittori e vengono inizializzati opportunamente i campi di ogni descrittore.

- **i descrittori sono frequentemente aggiornati dai vari programmi del NUCLEO**: di norma, ognuno di essi è considerato come una risorsa seriale che deve essere aggiornata in modo non interrompibile da parte del NUCLEO (vedi paragrafo successivo).

Un esempio può servire a rendere più intuitivo l’ultimo punto: il modo più semplice per codificare lo stato di un processo è di fare uso di apposite liste di descrittori di processo, una per ognuno dei vari stati riconosciuti dal NUCLEO.

Consideriamo ad esempio la lista dei processi eseguibili, ossia pronti ad essere eseguiti dalla CPU. Ogni qualvolta la funzione di fine quanto di tempo viene invocata, essa scandisce tale lista per inserire nella posizione appropriata il processo che ha terminato il suo quanto di tempo. Durante tale fase, il NUCLEO non deve essere interrotto finché il descrittore di processo non sia stato inserito. Nel caso opposto, infatti, la seconda attivazione del NUCLEO potrebbe richiedere l’esecuzione di un programma che consulta la stessa lista, e vi è la possibilità che essa sia stata lasciata in uno stato non coerente dalla precedente attivazione: ad esempio alcuni dei puntatori utilizzati per realizzare la lista potrebbero non essere stati ancora aggiornati.
9.8 Interrompibilità del NUCLEO

Per motivi di semplicità, l’interazione processo-NUCLEO è stata presentata come una attività sequenziale consistente nei seguenti tre passi:

1. il processo attiva il NUCLEO tramite int;
2. la CPU passa in stato Kernel ed il NUCLEO esegue gli opportuni programmi;
3. al termine, il NUCLEO rilascia l’uso della CPU ed il processo riprende l’esecuzione in stato User.

La Figura 9.2 illustra tale caso alquanto semplice.

La realtà è molto più complessa poiché il NUCLEO non deve soltanto soddisfare richieste emesse dal processo in esecuzione, ma anche gestire interruzioni di vario tipo quali quelle emesse dai dispositivi di I/O e quelle emesse dal chip interval timer.

In effetti, il NUCLEO non ha una struttura sequenziale come molti programmi, bensì parallela: ogni segnale d’interruzione riconosciuto dall’hardware e gestito dal sistema operativo causa l’attivazione di un apposito programma del NUCLEO. Tale struttura parallela assicura a sua volta la interrompibilità del NUCLEO, ossia la capacità di rispondere ad un segnale di interruzione mentre sta già trattando un altro segnale di interruzione (sia pure di tipo diverso).

Ogni NUCLEO interrompibile deve quindi essere in grado di eseguire in modo annidato le interruzioni, ossia deve essere in grado di sospendere l’esecuzione di
un programma di gestione di una interruzione per passare ad eseguire un altro programma di gestione di una interruzione di tipo diverso; terminata l’esecuzione del secondo programma, il NUCLEO deve riprendere l’esecuzione del primo programma.

La Figura 9.3 illustra un esempio di esecuzione annidata di tre programmi distinti del NUCLEO.

In effetti, l’attività più complessa nella progettazione di un NUCLEO consiste nel renderlo il più possibile interrompibile: in realtà, non sarà mai possibile renderlo totalmente interrompibile in quanto alcune operazioni critiche devono essere eseguite con le interruzioni disabilitate e con la certezza che il NUCLEO non verrà interrotto mentre esegue tali operazioni (vedi esempio precedente relativo all’inserimento di un descrittore di processo nella lista dei processi eseguibili).

Per esemplificare quanto appena detto, diamo alcuni esempi dei casi che possono verificarsi in seguito alla attivazione del NUCLEO tramite una qualche interruzione.

Nel caso più semplice considerato in precedenza, il programma del NUCLEO termina e riprende in stato User l’esecuzione del processo che ha eseguito la int (vedi Figura 9.2).

In altri casi, il programma del NUCLEO può richiedere una risorsa attualmente non disponibile, ad esempio, la lettura di un carattere da tastiera che non è stato ancora digitato dall’utente. Quando ciò si verifica, il NUCLEO provvede a porre il processo in stato di attesa ed a selezionare un altro processo da porre nello stato di esecuzione (vedi Figura 9.4).

In altri casi ancora, un processo in stato User Mode può avere generato una chiamata di sistema. Supponiamo che, mentre il NUCLEO sta gestendo la chiamata...

Figura 9.3: Esecuzione annidata di programmi del NUCLEO.
di sistema, si verifichi una interruzione di I/O di tipo \( j \) che fa passare un secondo processo nello stato di eseguibile. Supponiamo inoltre che il secondo processo abbia una priorità maggiore del primo. In questo caso, il NUCLEO può decidere di rimettere in esecuzione il secondo processo lasciando incompiuto il trattamento della eccezione (vedi Figura 9.5). Non tutti i NUCLEI tra quelli più diffusi sono in grado di effettuare tale tipo di “preemption”; quelli in grado di effettuarla sono chiamati \emph{NUCLEI preemptive}.

Gli esempi illustrati non sono in alcun modo esaurienti. Sono molto numerose infatti le possibili combinazioni di interazioni che devono essere considerate nel progettare un NUCLEO interrompibile e non è possibile elencarle tutte.
9.9 Processi di sistema

Abbiamo elencato nel paragrafo 9.2 alcune funzioni che debbono necessariamente essere realizzate tramite programmi del NUCLEO ma non abbiamo specificato se altre importanti funzioni del sistema operativo debbano essere realizzate all’interno del NUCLEO, oppure tramite appositi processi di sistema. Tali processi talvolta chiamati *demoni* sono invisibili all’utente nel senso che non interagiscono con il mouse oppure il monitor. Essi eseguono nel quarto Giga di spazio degli indirizzi comune a tutti i processi, ossia eseguono soltanto funzioni del kernel ed accedono soltanto a variabili del kernel.

Come tutti i processi, i processi di sistema possiedono un proprio PID, un proprio descrittore di processo e sono schedulati dal NUCLEO come i normali processi User Mode in base alla loro priorità.

Tutti i moderni sistemi operativi, a prescindere dalla loro impostazione (vedi prossima sezione), fanno un ampio uso di processi di sistema.

Unix, ad esempio, utilizza il demone *crond* per eseguire specifici compiti ad intervalli prefissati. Linux fa uso di un altro demone per verificare se vi è abbastanza RAM libera e, nel caso opposto, per recuperarla svuotando le cache software (page frame reclaiming).

9.10 Processo di sistema o NUCLEO?

Vi sono due scuole di pensiero riguardo al peso che devono avere i processi di sistema e quindi per complemento ai compiti attribuibili al NUCLEO.

Il primo approccio privilegia l’uso di processi di sistema per realizzare il maggior numero di funzioni possibili del sistema operativo, facendo uso di un NUCLEO ridotto chiamato *micro-NUCLEO* (in inglese, *microkernel*) che svolge pochi compiti essenziali quali la commutazione di processi ed una gestione semplificata delle interruzioni: nella maggior parte dei casi, il micro-NUCLEO riceve il segnale di interruzione ed invia un opportuno messaggio al processo interessato.

Le principali funzioni del sistema operativo vengono svolte da processi di sistema specializzati: esiste, ad esempio, un processo per lo scheduling, un processo per la gestione del file sistem, un processo per il controllo dei diritti d’accesso dei processi utente, un processo per la gestione delle aree di memoria RAM e così via.
Seguendo tale approccio, quando un processo utente intende richiedere un qualche servizio da parte del sistema operativo, esso non esegue una chiamata di sistema al NUCLEO ma invia un opportuno messaggio al processo di sistema gestore del servizio richiesto. In particolare, il processo richiedente esegue i seguenti passi:

1. esegue una chiamata di sistema di tipo `send()` (vedi paragrafo 7.3) al processo gestore specificando nel messaggio i parametri associati alla richiesta;

2. si pone in attesa del messaggio di risposta del gestore tramite una `receive()`.

I processi di sistema hanno una struttura ciclica: ognuno di essi usa una primitiva di sincronizzazione di tipo `receive()` per verificare se vi sono messaggi, ossia richieste di servizio; se vi è almeno un messaggio, passa ad eseguire la richiesta, invia un messaggio di risposta (vedi Figura 9.6) al processo richiedente e torna ad eseguire una `receive()`. In pratica, viene realizzata una architettura software di tipo “client/server” dove la maggior parte delle funzioni del sistema operativo sono svolte da processi di sistema di tipo server.

I vantaggi derivanti da tale impostazione sono numerosi:

- il sistema operativo risulta modulare e facile da mantenere ed aggiornare;
- la struttura del NUCLEO risulta estremamente semplificata; in particolare, scompare la necessità di rendere il NUCLEO interrompibile poiché esso è in grado di trattare ogni richiesta in tempi ridotti;
- l’uso di processi di sistema rende più agevoli i controlli sulla validità delle richieste; se la richiesta è di tipo gravoso, il server può decidere di creare più processi figli per poterla trattare in parallelo.
Purtroppo, esiste una grossa limitazione a tale approccio che è quello delle prestazioni: l’introduzione di numerosi processi di sistema fa aumentare notevolmente la comunicazione tra processi. Tale comunicazione realizzata tramite chiamate di sistema di tipo `send()` e `receive()` richiede frequenti interventi del NUCLEO ed induce overhead. I risultati sono quindi deludenti; oggi esistono pochi sistemi operativi commerciali basati sull’uso di micro-NUCLEI: il più noto è il sistema Next derivato dal prototipo Mach realizzato presso la Carnegie Mellon University.

Passiamo ora considerare un approccio diametralmente opposto a quello precedente: quello del NUCLEO monolitico che include la maggior parte delle funzionalità del sistema operativo all’interno del NUCLEO. In base a tale approccio, il NUCLEO include un elevato numero di programmi in grado di trattare decine o centinaia di chiamate di sistema diverse, che corrispondono ai diversi servizi offerti.

Esistono, ad esempio, chiamate per la gestione del file system, chiamate per la sincronizzazione di processi, chiamate per la richiesta di aree di memoria e così via. Alcune chiamate di sistema possono richiedere tempi di esecuzione elevati, per cui è indispensabile progettare il NUCLEO monolitico in modo interrompibile.

Non esistono molti punti a favore del NUCLEO monolitico se non quello cruciale dell’efficienza. Molti programmi del sistema operativo devono essere considerati alla stregua di programmi in tempo reale: possono essere eseguiti anche migliaia di volte al secondo ed il modo in cui vengono realizzati condiziona le prestazioni dell’intero sistema. Oggi, i più noti sistemi operativi sono realizzati facendo uso di NUCLEI monolitici. In alcuni casi, sono utilizzati un numero limitato di processi di sistema per alleggerire in parte la struttura del NUCLEO ma esso rimane comunque il componente del sistema operativo più complesso.

### 9.11 Thread o processi leggeri?

I processi descritti in precedenza sono una utile astrazione che consente di strutturare in modo ottimale il NUCLEO di un sistema multitasking. Ciò nonostante, il loro uso presenta alcuni inconvenienti dovuti alla ricchezza di informazioni associata ad ogni descrittore di processo.

In effetti, per effettuare una commutazione di processi, il NUCLEO deve salvare tutte le informazioni richieste nel descrittore del processo precedentemente in esecuzione e prelevare dal descrittore del nuovo processo le corrispondenti informazioni per registrarele nei registri del processore e nelle varie strutture di dati del sistema operativo. Tale operazione chiamata `commutazione di processi` (ro-
9.11. THREAD O PROCESSI LEGGERI?

ces switch) può risultare, nel caso di sistemi in tempo reale, eccessivamente costosa in termini di tempo di esecuzione. Per questo motivo, alcuni sistemi operativi fanno uso di un modello di esecuzione più complesso di quello considerato finora.

In tale modello, ogni processo è composto da uno o più thread. Ogni thread associato ad un processo condivide l’area di memoria assegnata al processo nonché i file da esso aperti.

Il descrittore di un singolo thread si limita quindi al cosiddetto “contesto di esecuzione”, ossia al contenuto dei registri del processore ed a quello dello stack usato dal thread. Lo scheduling (vedi paragrafo 6.6) non viene più effettuato tra processi, bensì tra thread ed, in generale, il NUCLEO tratta in modo paritetico i thread di uno stesso processo e quelli appartenenti a processi diversi quando deve selezionare il thread più adatto a cui assegnare la CPU.

Dal punto di vista realizzativo, è necessario per potere gestire thread introdurre appositi descrittori che si affiancano ai descrittori di processo; tali descrittori includono pochi campi, tra cui un identificatore di thread, un riferimento al processo a cui appartengono ed un “contesto di esecuzione”, ossia uno stack ed un’area di salvataggio dei registri della CPU.

Il tempo richiesto dal NUCLEO per effettuare la commutazione tra due thread risulta solitamente minore di quello richiesto per effettuare la commutazione tra due processi. I thread rappresentano quindi una valida scelta per realizzare applicazioni che condividono strutture di dati comuni e che sono eseguite su macchine dotate di più processori.

Linux non fa uso di thread ma utilizza un approccio diverso per ridurre il costo della commutazione di processi basato sui processi leggeri: tali processi condividono buona parte delle risorse possedute da un processo, tra cui lo spazio degli indirizzi e i descrittori di file aperti. La creazione di un processo leggero è più rapida di quella di un processo classico poiché è sufficiente copiare buona parte dei campi puntatore a risorse del processo padre in quelli del processo figlio.

La struttura del NUCLEO non risulta appesantita da un ulteriore tipo di descrittore ed è possibile realizzare applicazioni parallele basate su processi leggeri che non hanno nulla da invidiare, in termini di efficienza, a quelle basate sui thread. Attualmente, esistono NUCLEI che supportano i thread ed altri che supportano i processi leggeri e non sembrano esservi differenze significative in termini di prestazioni.
9.12 Quale linguaggio di programmazione?

La scelta dei linguaggi di programmazione utilizzati per realizzare il sistema operativo ha una sua importanza in quanto condiziona l’efficienza dell’intero sistema. Alcuni linguaggi di programmazione, infatti, includono dei meccanismi per la chiamata di procedure e per il controllo automatico degli indirizziamenti che rendono il codice macchina generato poco efficiente.

Come già sottolineato in precedenza, la parte critica del sistema operativo è costituita dal NUCLEO, mentre per le altre parti la scelta del linguaggio di programmazione non è determinante ai fini dell’efficienza del sistema. Per quanto riguarda la programmazione del NUCLEO, si assiste ad una evoluzione molto lenta verso linguaggi ad alto livello.

Fino agli anni ’70, tutti i NUCLEI erano codificati in linguaggio assemblativo: ad esempio, i NUCLEI dei sistemi operativi OS/360 e successori usati per i mainframe IBM e quello del sistema operativo VAX/VMS della Digital Equipment sono stati codificati interamente in linguaggio assemblativo.

È opportuno precisare che una piccola parte del NUCLEO, quella che riguarda le interazioni con l’hardware, deve necessariamente essere codificata nel linguaggio assemblativo per potere utilizzare le apposite istruzioni che operano sui registri della macchina.

Quando, ad esempio, il NUCLEO effettua una commutazione di processi, deve salvare il contenuto dei registri utilizzati dal processo sospeso in una apposita area di memoria (tipicamente, nel descrittore di processo e nello stack Kernel Mode del processo). Per fare ciò, serve un linguaggio di programmazione in grado di indirizzare specifici registri hardware della CPU, ossia serve il linguaggio assemblativo.

Si può stimare tra il 5 e il 10% la percentuale di codice del NUCLEO basata sull’architettura del calcolatore. La rimanente parte del NUCLEO tuttavia può essere scritta in un linguaggio ad alto livello, purché efficiente.

Unix è stato il primo sistema operativo che ha fatto un uso intenso di un linguaggio ad alto livello per codificare la maggior parte dei programmi del NUCLEO; in effetti, i progettisti di Unix hanno messo a punto un apposito linguaggio, il linguaggio C, per svolgere tale compito.

Dopo alcuni decenni, la situazione non è molto cambiata: molti NUCLEI qualifi quelli realizzati per i nuovi sistemi Unix continuano ad essere realizzati nel linguaggio C con alcune parti in linguaggio assemblativo. Una innovazione signi-
ficativa rispetto ai sistemi precedenti consiste nel separare in modo netto le parti del NUCLEO indipendenti dall’architettura da quelle dipendenti da essa. In tale modo risulta molto facilitata la portabilità del NUCLEO da una piattaforma hardware ad una altra.

Perfino un concetto innovativo come quello della programmazione ad oggetti che ha avuto un notevole impatto nella realizzazione di programmi applicativi ha incontrato un limitato successo tra i progettisti di NUCLEI, anche se alcuni sistemi operativi quali Windows XP fanno uso del linguaggio C++ per codificare alcune funzioni meno critiche del sistema operativo. Il motivo è sempre lo stesso, ossia l’inefficienza dei compilatori per linguaggi ad oggetto quale il C++ rispetto ai compilatori per linguaggi procedurali tradizionali.

L’unica piccola innovazione dal punto di vista delle tecniche di programmazione è consistita nel fare uso di un limitato numero di oggetti realizzati in C tramite strutture che contengono sia dati che puntatori a funzioni.

Un discorso a parte vale per quelle parti del sistema operativo realizzate tramite processi: poiché esse sono meno critiche dal punto di vista delle prestazioni, l’uso di linguaggi orientati ad oggetto può essere giustificato.
Capitolo 10

GESTIONE DELLA MEMORIA

10.1 Introduzione

Un compito importante del NUCLEO è quello di gestire le aree di memoria presenti nel calcolatore: la memoria principale (RAM) e la memoria secondaria (dischi di vario tipo).

Come è ovvio, la gestione della RAM è più critica di quella delle aree di disco, per cui le tecniche di gestione utilizzate sono alquanto diverse. Inoltre, le richieste di RAM da parte di funzioni del NUCLEO sono considerate prioritarie rispetto alle richieste di RAM effettuate da un processo in User Mode.

Per questi motivi, descriveremo in questo capitolo due classiche tecniche di gestione della memoria RAM utilizzate, rispettivamente, per soddisfare:

- assegnazione o rilascio di aree di RAM richieste da funzioni del NUCLEO;
- assegnazione di aree di RAM ad un processo in User Mode.

Descriveremo inoltre una tecnica di gestione dello spazio su disco utilizzata per soddisfare:

- richieste di modifica di dimensioni dei file emesse dal file system;
- richieste effettuate da altri componenti del NUCLEO per ottenere o rilasciare blocchi di disco.
10.2 Indirizzamento della RAM

Una premessa è d’obbligo prima di iniziare la discussione: gli accessi alla RAM da parte di un processo sono numerosissimi, dato che ogni nuova istruzione da eseguire va letta dalla RAM e che, in alcuni casi, l’esecuzione dell’istruzione richiede ulteriori accessi alla RAM per leggere o scrivere operandi. Per questo motivo, non è possibile pensare a schemi di gestione interamente software ma è necessario ricorrere ad appositi circuiti hardware che consentano determinati tipi di gestione senza aumentare in modo significativo il tempo d’accesso alla RAM che è oggi dell’ordine di poche decine di miliardesimi di secondo.

Un’altra importante considerazione è che la gestione multitasking, più volte citata in precedenza, richiede l’uso di una tecnica di suddivisione della memoria RAM. Infatti, tale tipo di gestione presuppone che, ad ogni istante, vi siano più programmi indipendenti caricati in memoria.

Poiché i programmi hanno tempi di esecuzione diversi e lunghezze diverse, nasce il problema di definire tecniche di indirizzamento della RAM che consentano da un lato di caricare agevolmente nuovi programmi in memoria e, dall’altro, di utilizzare efficientemente la memoria a disposizione.

10.2.1 Paginazione

La paginazione è una tecnica di indirizzamento della RAM che fa uso di sofisticati circuiti di traduzione degli indirizzi. Nel descrivere tale tecnica, faremo uso della seguente terminologia:

- **indirizzo logico**: numero da 32 o 64 bit usato nelle istruzioni in linguaggio macchina per identificare l’indirizzo della prossima istruzione da eseguire, oppure l’indirizzo di un operando richiesto dall’istruzione;

- **indirizzo fisico**: numero da 32 o 48 bit usato dal bus per indirizzare byte di memoria RAM contenuta in appositi chip.

Ogni indirizzo logico è tradotto dal circuito di pagina in un indirizzo fisico. Come vedremo appresso, indirizzi logici contigui possono essere tradotti in indirizzi fisici non contigui. La memoria RAM disponibile è suddivisa in N blocchi (in inglese, *page frame*) di lunghezza fissa, ad esempio 4096 byte (sono anche usati blocchi di dimensioni maggiori).
L’informazione contenuta in uno di tali blocchi prende il nome di *pagina*. Per semplificazione, descriveremo il meccanismo di paginazione per CPU utilizzano indirizzi logici da 32 bit. In tale caso, ogni indirizzo logico è decomposto in un numero di pagina (i 20 bit più significativi) e un indirizzo relativo nella pagina (i rimanenti 12 bit).

Per ridurre la dimensione delle tabelle, i 20 bit che esprimono il numero di pagina sono decomposti ulteriormente in due gruppi di 10 bit: il primo gruppo indica la posizione nella tabella principale chiamata *Page Directory*, mentre il secondo gruppo indica la posizione in una delle tabelle secondarie chiamate *Page Table*.

La trasformazione dell’indirizzo logico in un indirizzo fisico è effettuata dal circuito di paginazione nel seguente modo (vedi Figura 10.1):

1. ottieni da un apposito registro della CPU l’indirizzo della PageDirectory del processo in esecuzione (nei microprocessori Intel 80x86, tale registro è chiamato *cr3*);
2. usa i 10 bit più significativi dell’indirizzo logico per selezionare la voce opportuna nella Page Directory;
3. leggi il contenuto della voce selezionata al passo precedente per indirizzare la Page Table contenente la pagina da indirizzare;
4. usa i 10 bit intermedi dell’indirizzo logico per selezionare la voce opportuna della Page Table;
5. somma i 12 bit meno significativi dell’indirizzo logico all’indirizzo fisico della page frame di RAM ottenuta al passo precedente ottenendo così l’indirizzo fisico desiderato.

Ogni voce di una Page Directory o di una Page Table occupa 32 bit di cui:

- 20 bit contengono l’indirizzo fisico di una page frame (poiché le page frame sono lunghe $2^{12} = 4096$ byte, non è necessario specificare i 12 bit meno significativi dell’indirizzo fisico);
- 12 bit contengono diversi flag che caratterizzano lo stato della page frame ed i suoi diritti d’accesso. Tipicamente, sono usati i seguenti flag:
  - flag Present: vale 0 se la pagina non è attualmente presente in RAM;
  - flag User/Supervisor: vale 0 se la pagina può essere indirizzata solo quando la CPU è in Kernel Mode;
Figura 10.1: Trasformazione di un indirizzo logico in un indirizzo fisico.

- flag Execute: vale 1 se la CPU è autorizzata a prelevare istruzioni dalla pagina;
- flag Read: vale 1 se la CPU è autorizzata a leggere dati dalla pagina;
- flag Write: vale 1 se la CPU è autorizzata a scrivere dati nella pagina.

Se il flag Present vale 0, oppure se i diritti d’accesso non consentono l’indirizzamento richiesto dalla CPU, il circuito di paginazione genera una eccezione di tipo Page Fault che attiva un apposito programma del NUCLEO.

Nell’accedere ad una voce di Page Directory o di Page Table, il circuito di paginazione verifica inoltre se la voce è nulla (tutti i 32 bit hanno il valore 0). Anche in questo caso, il circuito di paginazione genera una eccezione di tipo Page Fault.

Prima di generare una eccezione, il circuito di paginazione salva in un apposito registro (registro cr4 nel caso dei microprocessori Intel 80x86) l’indirizzo logico dell’operando che ha causato l’eccezione. Tale indirizzo è diverso da quello dell’istruzione che lo contiene. Quest’ultimo indirizzo è contenuto nel registro eip ed è salvato automaticamente dall’hardware nell’stack Kernel Mode.

Potrebbe sembrare a prima vista che l’uso della paginazione degrada sensibilmente le prestazioni del calcolatore: ogni accesso da parte della CPU ad un dato oppure ad una istruzione contenuti nella RAM richiede tre indirizzamenti consecutivi alla RAM anziché uno solo!
In pratica, sono usati vari accorgimenti per ridurre il tempo di consultazione delle tabelle di traduzione. Oggi tutti i chip CPU includono memorie cache veloci di dimensioni ridotte in cui vengono conservate le coppie di valori (indirizzo logico, voce di Page Table o Page Directory) degli ultimi indirizzamenti effettuati. Grazie a tali cache, non è necessario accedere alla RAM per leggere una voce di Page Directory o di Page Table in quanto tale informazione è già presente nella cache, per cui il tempo medio di accesso alla RAM non si scosta significativamente da quello richiesto per effettuare un singolo accesso.

È importante sottolineare come la paginazione consenta di proteggere efficacemente gli spazi degli indirizzi dei vari processi. In pratica, ogni processo può accedere soltanto alle page frame che sono state ad esso assegnate dal sistema operativo. D’altra parte, un processo User Mode non può modificare le proprie tabelle di paginazione e quindi modificare il proprio spazio degli indirizzi. In effetti, le tabelle di paginazione dei vari processi sono contenute nelle page frame riservate al NUCLEO (flag User/Supervisor impostato a 0) e la protezione è quindi assicurata.

### 10.2.2 Indirizzi logici riservati ai programmi del NUCLEO

Ogni processo possiede una propria Page Directory ed un gruppo di Page Table. Grazie a tale approccio, gli stessi indirizzi logici possono essere usati da più processi senza che ciò causi alcuna ambiguità poiché la traduzione da indirizzo logico ad indirizzo fisico avviene utilizzando le tabelle di paginazione del processo che richiede l’indirizzamento.

Il NUCLEO invece non è un processo e sarebbe alquanto macchinoso assegnare ad esso apposite tabelle di paginazione. La soluzione preferita consiste invece nel riservare un parte degli indirizzi logici per i programmi del NUCLEO ed includere nelle tabelle di paginazione di ogni processo un mapping tra gli indirizzi logici riservati al NUCLEO e gli indirizzi fisici della RAM installata.

Vediamo come ciò viene realizzato su un esempio concreto. In Linux, l’intervallo di 4 Giga (\(2^{32}\)) corrisponde a tutti i possibili indirizzi logici viene suddiviso in due intervalli:

- un intervallo da 0 a 3 Giga meno 1 (\(3 \times 2^{30} - 1\)) riservato ai processi che eseguono in User Mode;
- un intervallo da 3 Giga a 4 Giga meno 1 (\(4 \times 2^{30} - 1\)) riservato ai processi che eseguono in Kernel Mode, ossia ai programmi del NUCLEO.
Poiché ogni Page Directory include 1024 voci, ciò significa che le prime 768 voci sono riservate ai processi che eseguono in User Mode, mentre le ultime 256 sono riservate ai programmi del NUCLEO.

Ogni processo (tranne quelli che condividono lo spazio degli indirizzi con altri processi) possiedono un proprio gruppo di tabelle di paginazione. Le ultime 256 voci della Page Directory sono però identiche per tutti i processi.

In Linux il numero di indirizzi logici a disposizione del NUCLEO consente di mappare fino a 896 MB di RAM. Se la quantità di RAM installata è inferiore a tale valore alcune delle ultime 256 voci della Page Directory sono nulle per indicare che i corrispondenti indirizzi logici non sono validi in quanto non vi è RAM associata ad essi.

È importante osservare che il mapping tra indirizzi logici usati dal NUCLEO e indirizzi fisici è una semplice trasformazione lineare:

\[
\text{indirizzo fisico} = \text{indirizzo logico} - 0xc0000000
\]

Ovviamente tale mapping vale solo per i programmi del NUCLEO e non per i processi in User Mode. Come vedremo nel prossimo paragrafo, il NUCLEO utilizza un algoritmo di gestione della memoria che assegna aree contigue di RAM, per cui non è necessario modificare il contenuto di alcuna delle ultime 256 voci della Page Directory in seguito ad una assegnazione o rilascio di memoria.

### 10.3 L’algoritmo Buddy System

Vediamo ora come il NUCLEO gestisce la RAM facendo uso dei circuiti di paginazione. Come è ovvio, l’unità minima di allocazione è la page frame, ossia 4096 byte.

In primo luogo, il NUCLEO deve riservare, durante la fase di inizializzazione, un sufficiente numero di page frame per contenere il codice e le strutture di dati statiche. Le page frame rimanenti prendono il nome di *memoria dinamica* e possono essere allocate e rilasciate dinamicamente per soddisfare le esigenze delle varie funzioni del NUCLEO.

In effetti, oltre a poche strutture statiche di dati, il NUCLEO fa uso di numerose strutture dinamiche di dati che vengono create e successivamente eliminate. Diamo alcuni esempi di tali strutture:
10.3. L’ALGORITMO BUDDY SYSTEM

- descrittore di processo: quando viene creato un nuovo processo, è necessario assegnare una nuova area di memoria per contenere il descrittore di processo; quando il processo viene eliminato, tale area di memoria può essere riutilizzata;

- inode: quando viene aperto un file per la prima volta, il file system copia da disco il descrittore del file in una struttura chiamata inode; l’area di memoria usata da un inode può essere rilasciata quando non vi sono più processi che hanno aperto quel file;

- messaggi: le API che consentono di inviare messaggi, ad esempio la `msgsnd` della suite Inter Process Communication (IPC), richiedono aree di memoria per contenere i messaggi pendenti destinati ai vari processi; tali aree di memoria possono essere rilasciate quando i processi destinatari hanno “consumato” i messaggi.

Tipicamente, il NUCLEO centralizza la gestione della RAM dinamica tramite due funzioni del tipo `get_mem()` e `free_mem()` che possono essere usate dalle varie funzioni del NUCLEO per ottenere o rilasciare page frame.

Per evitare problemi di frammentazione della memoria libera, è molto usato l’algoritmo Buddy System che tenta di accorpare blocchi contigui di page frame libere creando così un singolo blocco libero di dimensioni maggiori. Vediamo in dettaglio come opera tale algoritmo.

L’algoritmo gestisce blocchi di page frame di dimensioni diverse, ad esempio blocchi da 1, 2, 4, 8, 16, 32, 64, ..., 512 page frame, per cui la funzione `get_mem()` prevede un parametro che specifica la dimensione del blocco richiesto.

Per ogni tipo di blocco, il Buddy System gestisce un apposito descrittore che specifica gli indirizzi dei blocchi liberi e quelli dei blocchi occupati. Se consideriamo ad esempio i blocchi da 64 page frame, l’intera RAM verrà vista come un vettore di bit dove ogni bit denota 64 × 4 KB di RAM; il bit vale 0 se il blocco è libero, 1 se è occupato.

Nel soddisfare una richiesta per un blocco di dimensione \( k \), l’algoritmo Buddy System consulta dapprima il descrittore per blocchi di dimensione \( k \): se vi è almeno un bit uguale a 0, pone tale bit a 1 e ritorna l’indirizzo iniziale del blocco appena individuato. Se non vi sono bit uguali a 0, consulta il descrittore per blocchi di dimensione \( 2 \times k \) e così via fino a trovare il primo descrittore che include un blocco libero.

Se non vi è alcun descrittore contenente un bit uguale a 0, la richiesta di memoria non può essere soddisfatta e la funzione `get_mem()` ritorna un codice di errore.
Altrimenti, viene selezionato il primo descrittore avente un bit uguale a 0 ed avviene la suddivisione del blocco in un blocco occupato ed uno o più blocchi liberi.

Facciamo un esempio specifico per meglio chiarire come ciò si verifica. Supponiamo che la richiesta è per 64 page frame e che il blocco libero più piccolo occupa 256 page frame. In questo caso, il blocco libero da 256 page frame viene dichiarato occupato e viene suddiviso in un blocco occupato da 64 page frame, un blocco libero da 64 page frame ed un blocco libero da 128 page frame. I relativi descrittori di blocchi da 64, 128 e 256 page frame vengono opportunamente modificati.

Nel rilasciare un blocco, l'algoritmo Buddy System verifica se il blocco "gemello" (in inglese “buddy”) di quello che si è appena liberato è anch’esso libero. Nel caso affermativo, fonde i due blocchi liberi buddy in un unico blocco libero di dimensione doppia rispetto a quello del blocco che si è reso libero.

Anche in questo caso, facciamo un esempio specifico per meglio chiarire cosa si intende per coppia di blocchi buddy. Consideriamo i blocchi da 16 K, ossia i blocchi da 4 page frame. Gli indirizzi fisici associati a tali blocchi sono multipli di 16 K: 0, 16, 32, 48, 64, 80, . . . I blocchi buddy di tale gruppo sono 0 con 16, 32 con 48, 64 con 80 e così via. In effetti, fondendo il blocco 64 con quello 80 si ottiene un blocco da 32 K avente indirizzo iniziale multiplo di 32. Viceversa, due blocchi contigui da 16 K quali il blocco 48 e quello 64 non sono buddy perché l'indirizzo iniziale del blocco ottenuto fondendo tale coppia di blocchi non è un multiplo di 32 K.

10.4 Estensioni del Buddy System

Un limite dell’algoritmo Buddy System appena illustrato è che l’unità minima di allocazione di memoria è un page frame, ossia 4 KB. Se le richieste di memoria delle funzioni del NUCLEO sono limitate (poche decine o centinaia di byte), assegnare un intera page frame quando basterebbe una piccola frazione di essa costituisce uno spreco di memoria.

I questi casi, è possibile realizzare un sistema di cache software che include blocchi liberi da 32, 64, 128, 256, 512, 1024, 2048 byte e soddisfare le richieste delle funzioni del NUCLEO cercando il blocco libero avente dimensione maggiore o uguale a quello del numero di byte richiesti: se, ad esempio, la funzione richiede 100 byte otterrà un blocco da 128 byte.
10.5 Demand Paging

Il NUCLEO è un componente privilegiato del sistema operativo, per cui le funzioni del NUCLEO ottengono direttamente tutta la memoria dinamica richiesta. Inoltre, per non modificare le ultime 128 voci della Page Global Directory che devono risultare uguali per tutti i processi, il NUCLEO ottiene sempre aree contigue di memoria libera.

Del tutto diversa è l’assegnazione di memoria ai processi. In questo caso vale la strategia opposta: l’assegnazione di RAM ad un processo viene ritardata il più a lungo possibile, ossia fino a quando il processo esegue una istruzione che necessita la presenza di una area di memoria non ancora allocata. Tale strategia prende il nome di demand paging (paginazione a richiesta) e presuppone l’esistenza nel NUCLEO di appositi descrittori che identificano le regioni di memoria possedute dal processo (vedi capitolo 8).

A differenza del NUCLEO, un processo ottiene una page frame libera alla volta e le page frame ottenute non hanno indirizzi fisici contigui. In altre parole, ogni assegnazione di RAM ad un processo implica una modifica di una Page Table del processo.

Vediamo ora in che modo il NUCLEO è in grado di realizzare il demand paging.

Quando un processo inizia ad eseguire istruzioni, esso non ha ancora ottenuto alcuna page frame di RAM da parte del NUCLEO. Ha invece ottenuto diverse regioni di memoria, per cui prima di iniziare l’esecuzione del processo, il NUCLEO provvede ad inizializzare opportunamente i descrittori di memoria del processo.

Alcune regioni di memoria mappano (vedi paragrafo 8.5) porzioni del file eseguibile associato al processo: la regione codice mappa una porzione del file eseguibile, la regione dati inizializzati mappa un’altra porzione del file eseguibile e così via. L’indicazione della porzione di file mappata è registrata nel descrittore di regione.
Altre regioni di memoria, ad esempio la regione stack non mappano alcuna regione di file ma costituiscono un mapping anonimo, per cui quando il processo tenterà di accedere ad un indirizzo incluso in una di tali regioni sarà sufficiente assegnare ad esso una page frame di RAM dinamica senza doverla inizializzare con dati o codice letti dal file eseguibile.

Continuiamo con la descrizione di cosa avviene quando un processo ottiene per la prima volta l’uso della CPU. Nell’eseguire la prima istruzione avente un determinato indirizzo logico, il circuito di paginazione indirizzerà la voce opportuna della Page Directory e la troverà posta a 0. In conseguenza, genererà una eccezione di tipo “errore di pagina” ed un apposito programma del NUCLEO chiamato “gestore degli errori di pagina” prenderà il controllo (vedi paragrafo 9.3). A questo punto il gestore verificherà che la pagina appartiene effettivamente alla regione codice del processo, per cui eseguirà i seguenti passi:

1. ottieni una page frame libera;
2. copia in essa la porzione di codice contenuta nel file eseguibile (la regione codice è mappata in una parte del file eseguibile);
3. modifica la Page Table inserendo in essa l’indirizzo fisico della page frame ottenuta;
4. riesegui l’istruzione che ha causato l’errore di pagina.

In generale, quando si verifica una eccezione di tipo “errore di pagina”, il gestore distingue i seguenti casi possibili e, a seconda del caso, prende gli opportuni provvedimenti:

- l’indirizzo logico che ha causato l’eccezione non appartiene allo spazio degli indirizzi del processo: in questo caso, il NUCLEO elimina il processo che ha causato l’eccezione;
- l’indirizzo logico che ha causato l’eccezione appartiene allo spazio degli indirizzi del processo ma il processo non è autorizzato ad accedere alla page frame nel modo richiesto (tentativo di scrittura su pagine di sola lettura, ecc.): anche in questo caso, il NUCLEO elimina il processo che ha causato l’eccezione;
- l’indirizzo logico che ha causato l’eccezione appartiene allo spazio degli indirizzi del processo ed il processo è autorizzato ad accedere alla page frame nel modo richiesto: in questo caso va attivato il demand paging che viene realizzato in modo diverso a seconda che il mapping della regione coinvolta sia di tipo anonimo oppure sia associato ad una porzione di file.
10.6. RECUPERO DI MEMORIA LIBERA

– mapping anonimo: il gestore invoca get_mem() per ottenere una page frame libera ed inserisce l’indirizzo fisico di tale page frame nei 20 bit più significativi della voce della Page Directory o Page Table. I flag della voce sono impostati in base ai valori associati al descrittore della regione di memoria; a questo punto l’eccezione è stata trattata e l’esecuzione del processo può riprendere;

– mapping di una porzione di file: oltre ad eseguire le azioni svolte per il mapping anonimo, il gestore provvede ad iniziare una lettura di 4 KB della porzione di file mappata e pone il processo nello stato di “attesa della fine lettura dati da disco”; in questo caso, l’esecuzione del processo potrà riprendere solo dopo la fine della lettura dei dati richiesti da disco.

• pagina swappata: un apposito flag (flag Present = 0) incluso nella voce della Page Table indica che la pagina è attualmente salvata in un’area di swap del disco (vedi prossimo paragrafo); in questo caso, il gestore provvede a ricaricare la pagina dall’area di swap e rimette il processo in esecuzione; durante la fase di swap-in in il processo viene posto nello stato di bloccato.

10.6 Recupero di memoria libera

Nei lontani anni ’70, per supplire alla limitata quantità di RAM disponibile, fu inventato uno schema di gestione chiamato memoria virtuale in cui un’area di disco veniva considerata come una estensione logica della RAM. Tale schema, interamente trasparente per i programmatori, consentiva di mandare in esecuzione programmi che richiedevano più RAM di quella effettivamente disponibile. Oggi la memoria virtuale è caduta in disuso data la notevole capacità raggiunta dalle attuali RAM.

A prescindere dalla dimensione della RAM, vi sono comunque casi in cui, vuoi per l’elevato numero di processi, vuoi per l’abbondante uso di RAM effettuato da alcuni processi, l’intera RAM viene saturata ed il NUCLEO si trova a corto di page frame libere. In tali casi viene invocato un algoritmo euristico di recupero di memoria libera o page frame reclaiming per recuperare memoria. Tale algoritmo recupera in primo luogo pagine incluse in cache che non sono più usate da alcun processo. Se ciò non è sufficiente, toglie pagine ai processi e, se necessario ne ricopia il contenuto su disco.

Le pagine considerate dall’algoritmo sono le seguenti:
• pagine di disk cache (tali pagine sono gestite dal NUCLEO e servono soltanto a ridurre il numero di accessi al disco);

• pagine di una regione di memoria di un processo che ha una immagine su disco (file mappato in memoria);

• pagine di una regione di memoria di un processo che non ha una immagine su disco (mapping anonimo).

Nel primo caso, la pagina è conservata in RAM per possibili usi futuri ma non è attualmente usata da alcun processo, per cui la page frame che la contiene può essere immediatamente recuperata.

Nel secondo caso, la pagina va prima salvata su disco nel file di appartenenza e successivamente la page frame che la contiene può essere recuperata.

Nel terzo caso, non esiste un file dove salvare la pagina che contiene una porzione dello stack oppure una porzione dello heap. D’altra parte, sarebbe penalizzante per l’algoritmo di page frame reclaiming non potere togliere pagine di stack o pagine di memoria dinamica ad un processo che ne fa un uso sovrabbondante. Per questo motivo viene introdotta una apposita partizione su disco chiamata partizione di swap in cui salvare, se richiesto, le pagine di mapping anonimi usate dai vari processi.

Per quanto riguarda le tabelle di paginazione, una pagina appartenente al processo ma swappata su disco viene identificata impostando i 32 bit della voce di Page Table nel seguente modo:

• il flag di Present viene posto a 0;

• i rimanenti bit sono usati per specificare l’indirizzo nell’area di swap dove è registrata la pagina.

Quando il processo tenta di accedere ad una pagina swappata su disco, il gestore delle eccezioni di page fault riconosce, dal fatto che la voce della Page Table non è nulla ma che il flag Present è uguale a 0, che la pagina richiesta è stata swappata su disco. In questo caso, ottiene una page frame libera ed avvia una procedura di swap in per leggere la pagina da disco.
10.7 Gestione dello spazio su disco

Anche lo spazio su disco è una risorsa che deve essere gestita opportunamente. Il file system è il principale interessato nell’ottenere blocchi di disco liberi per potere ampliare la dimensione di un file. In modo analogo, il file system rilascia blocchi di disco quando cancella un file o quando ne riduce la dimensione.

A differenza della RAM, la gestione dello spazio su disco non richiede appositi circuiti hardware quali l’unità di paginazione ma può essere interamente realizzata in software da appositi programmi del NUCLEO. In effetti, i tempi di indirizzamento del disco sono molto elevati per cui il peso relativo dei programmi per la gestione dello spazio su disco risulta trascurabile.

La gestione dello spazio su disco è realizzata in modo alquanto semplice considerando il disco come un vettore di  *blocchi fisici* aventi numeri d’ordine 0, 1, 2, .... Ogni blocco ha una dimensione fissa, tipicamente 1024, 2048 o 4096 B.

Il file system usa un’altra grandezza per identificare i blocchi che compongono un file: ogni file è considerato come un vettore di  *blocchi di file* aventi numeri d’ordine 0, 1, 2, .... Ogni blocco di file ha la stessa dimensione di un blocco fisico. Espandere un file significa ottenere ulteriori blocchi di file; troncare un file significa eliminare blocchi di file a partire dall’ultimo.

Si noti l’analogia tra indirizzo fisico di un page frame e blocco fisico da un lato, e tra indirizzo logico di una pagina di dati e blocco di file dall’altro. Come per la gestione della RAM, ogni descrittore di file include una apposita struttura che associa ai blocchi di file i corrispondenti blocchi fisici. In questo modo, il file system può effettuare indirizzamenti all’interno del file facendo riferimento a blocchi di file e può quindi ignorare quali specifici blocchi fisici sono stati assegnati al file.

In conclusione, la gestione dello spazio consiste quindi in una semplice paginazione realizzata in software che consente di:

- sfruttare al meglio lo spazio su disco (un semplice vettore di bit identifica i blocchi fisici liberi e quelli occupati);
- consentire di espandere o contrarre la dimensione dei file in modo arbitrario, senza dovere allocare i dati di un file in blocchi fisici contigui.
Capitolo 11

GESTIONE DEI DISPOSITIVI DI I/O

11.1 Introduzione

L’esistenza di numerosi dispositivi di I/O, ognuno con caratteristiche particolari, complica notevolmente la struttura del NUCLEO che deve includere appositi programmi di gestione per ognuno dei dispositivi “supportati”. Nei NUCLEI attuali, oltre il 50% del codice è dedicato alla gestione dei dispositivi di I/O.

Data la complessità dell’argomento, la trattazione svolta in questo capitolo risulterà necessariamente sommaria. Per concretaezza, faremo alcuni riferimenti al modo in cui il sistema operativo Unix gestisce le operazioni di I/O.

11.2 Architettura di I/O

Prima di iniziare a descrivere il modo in cui il NUCLEO gestisce i dispositivi di I/O, è opportuno dare alcuni cenni sulla architettura di I/O, ossia sul modo in cui i dispositivi di I/O sono collegati agli altri componenti dell’elaboratore.

Come indicato nella Figura 11.1, il dispositivo di I/O (stampante, disco, tastiera, mouse, ecc.) è collegato ad un apposito controllore hardware. Tale controllore è collegato, a sua volta, ad una interfaccia che consente di inviare comandi al dispositivo, leggerne lo stato attuale e trasferire dati tramite appositi buffer. In altre parole, il dispositivo deve includere una memoria locale diversa dalla
RAM del sistema e tale memoria locale deve essere composta da diversi registri indirizzabili sia dalla CPU che dal controllore del dispositivo.

Nelle architetture x86 e x86_64 tali registri possono essere di due tipi:

- porte di I/O
- registri di I/O

Per accedere ad una porta di I/O la CPU usa apposite istruzioni chiamate \texttt{in} e \texttt{out} ed utilizza appositi indirizzi fisici da 16 bit, per cui ogni porta di I/O ha un indirizzo fisico compreso tra 0 e 0x7ffff.

Per accedere ad un registro di I/O, la CPU usa normali istruzioni di tipo \texttt{mov}. Gli indirizzi fisici utilizzati in questo caso sono indirizzi fisici speciali, diversi da quelli utilizzati per accedere a celle di RAM. Per evitare confusione, li indicheremo col termine \textit{indirizzi fisici di I/O}. Tali indirizzi, tuttavia, non possono comparire esplicitamente come operandi di una istruzione di tipo \texttt{mov}. Infatti gli operandi delle istruzioni sono sempre indirizzi logici. Soltanto quando l’istruzione è eseguita, i circuiti del Memory Management Unit (MMU) si avvalgono delle tabelle di paginazione per tradurre l’indirizzo logico in un indirizzo fisico.

Per accedere ai registri di I/O di un dispositivo, è quindi necessario avere effettuato prima una operazione chiamata \textit{I/O remapping}. Tale operazione consiste, in sintesi, nel modificare alcune Page Table riservate al kernel in modo da riservare appositi intervalli di indirizzi logici per ogni dispositivo di I/O che fa uso di registri di I/O e nel far corrispondere a tali indirizzi logici i corrispondenti indirizzi fisici assegnati dal BIOS ai vari dispositivi. Grazie a tale accorgimento, le istruzioni di tipo \texttt{mov} che accedono a registri di I/O possono fare uso di indirizzi logici.

Il proc file \texttt{/proc/iomem} consente di visualizzare gli indirizzi fisici di I/O assegnati ai vari dispositivi.

L’I/O remapping è eseguito durante la inizializzazione del kernel, oppure quando viene fatto un hotplug del dispositivo.

Leggendo da o scrivendo in porte o registri di I/O è possibile realizzare le seguenti funzioni:

- inviare un comando al dispositivo;
- leggere lo stato del dispositivo;
11.2. ARCHITETTURA DI I/O

La maggior parte dei dispositivi di I/O prevede la possibilità di emettere appositi segnali di interruzione per segnalare la fine di una operazione di I/O. Esiste inoltre un apposito flag, programmabile tramite porta di I/O, che abilita o disabilita le interruzioni di I/O emesse dal dispositivo.

Dispositivi con elevato tasso di trasferimento (ad esempio, i dischi magnetici) possono essere collegati ad un processore autonomo di I/O chiamato Direct Memory Access Controller (DMAC). Anche in questo caso, è possibile programmare tramite appositi flag di una porta di I/O l’attivazione del DMA per il dispositivo. Grazie al DMAC, il NUCLEO è in grado di avviare una operazione di I/O relativamente complessa, ad esempio la lettura di più blocchi di dati da disco in un’area prefissata di RAM, per poi passare a svolgere altre attività senza aspettare la terminazione dell’operazione di I/O. Quando il DMAC ha terminato di trasferire dati, invia un segnale alla interfaccia di I/O interessata, la quale genera, a sua volta, un segnale di interruzione.
11.3 Dispositivi di I/O riconosciuti dal file system

Ogni sistema operativo tenta di “virtualizzare” nel modo più efficace possibile i vari dispositivi di I/O supportati offrendo al programmatore una interfaccia omogenea.

Diamo alcuni cenni sul come ciò viene realizzato nel sistema operativo Unix. In primo luogo, Unix riconosce due tipi di dispositivi di I/O:

- dispositivi a caratteri: sono dispositivi lenti (tastiera, stampante, ecc.) che trasferiscono dati un carattere alla volta e non richiedono buffer, oppure dispositivi veloci che fanno uso di buffer (schede video, schede audio, ecc.) ma che usano una sola volta i dati contenuti nel buffer in quanto gli accessi sono di tipo sequenziale;

- dispositivi a blocchi: sono dispositivi più complessi (dischi di vario tipo) che trasferiscono dati un blocco alla volta e richiedono pertanto un buffer per contenere tale blocco. Inoltre, sono in grado di indirizzare il dispositivo in modo casuale, per cui lo stesso blocco può essere riutilizzato in istanti diversi.

Ogni dispositivo è identificato da Unix tramite due identificatori chiamati major number e minor number. Il major number identifica il tipo di dispositivo. Il minor number è usato per identificare uno specifico dispositivo appartenente ad un gruppo di dispositivo tra loro omogenei (ad esempio, i vari dischi collegati ad uno stesso controllore).

Ogni dispositivo è rappresentato dal file system di Unix tramite un apposito device file di tipo orientato a blocchi oppure orientato a caratteri (vedi Paragrafo 4.4.3).

11.3.1 Programmazione di un device file

In Unix, la programmazione ad alto livello di un device file non differisce da quella di un file standard: le stesse “file operation” open(), close(), read(), write(), lseek(), ecc. usate sui file standard possono essere usate su device file. Ad esse si aggiunge la chiamata di sistema ioctl() che consente di eseguire operazioni specifiche al dispositivo (ad esempio, aprire il cassetto di un lettore di CD).
11.3. DISPOSITIVI DI I/O RICONOSCIUTI DAL FILE SYSTEM  

Il programma illustrato appresso (si tratta essenzialmente di una versione semplificata del comando Unix `cp`) mette in evidenza la versatilità del file system Unix: il programma di copiatura tra file è in grado di operare sia su device file che su file standard e prescinde dal file system in cui sono registrati i file standard (Ext2, VFAT, ecc.):

```c
inf = open(p, O_RDONLY, 0);
outf = open(q, O_WRONLY|O_CREAT|O_TRUNC, 0644);
  do {
    n = read(inf, buf, 4096);
    write(outf, buf, n);
  } while (n);
  close(outf);
  close(inf);
```

Ovviamente, vi sono limitazioni legate al tipo di dispositivo di I/O, per cui ogni dispositivo ammette un sottoinsieme di tutte le file operation esistenti.

Se il dispositivo è un dispositivo di solo input, ad esempio la tastiera, la file operation `write()` non è applicabile a tale dispositivo. In modo analogo, se il dispositivo è di tipo orientato a caratteri, la file operation `lseek()` non è applicabile a tale dispositivo.

Solitamente, tutti i device file sono inclusi nella directory `/dev`. Per potere includere un nuovo dispositivo di I/O in tale directory, è necessario avere prima “registrato” il corrispondente driver. Con tale termine si intende l’attività mediante la quale il sistema operativo acquisisce informazioni circa un nuovo tipo di driver e sui vari dispositivi dello stesso tipo che esso è in grado di gestire. I principali dati passati nella fase di registrazione riguardano:

- major number del dispositivo;

- funzioni specializzate che realizzano il gruppo di file operation applicabili al dispositivo (se una funzione non è applicabile, viene passato un puntatore nullo).

La chiamata di sistema `mknod()` consente di creare un device file. I parametri utilizzati sono il nome del device file, il tipo (a blocchi oppure a caratteri) nonché il major ed il minor number.
11.4 Supporto del NUCLEO

Vediamo ora in che modo il NUCLEO supporta la gestione dei dispositivi di I/O. A seconda del tipo di dispositivo, il NUCLEO può offrire un supporto limitato alla porta di I/O oppure un supporto totale.

Alcune applicazioni, ad esempio quelle che realizzano videogiochi, possono rinunciare per motivi di efficienza ad usufruire delle chiamate di sistema offerte dal NUCLEO ed accedere direttamente alla scheda grafica. In tale caso, le applicazioni fanno uso della chiamata di sistema `ioperm()` per potere accedere alle porte di I/O utilizzate.

Un esempio relativo al supporto limitato alla porta di I/O riguarda le porte seriali presenti nel calcolatore: il NUCLEO supporta le porte seriali presenti (`/dev/ttyS0`, `/dev/ttyS1`, ecc.) consentendo al programmatore di usare le file operation `read()` e `write()` su tali porte considerate come dei device file.

Il NUCLEO non supporta invece alcuno dei vari dispositivi collegabili alla porta seriale (modem, mouse, stampante, ecc.). La gestione di tali dispositivi deve essere effettuata da appositi programmi applicativi esterni al NUCLEO.

Il supporto totale è il più impegnativo in quanto richiede la realizzazione di un apposito programma chiamato gestore del dispositivo di I/O (in inglese I/O `driver`) per ognuno dei dispositivi riconosciuti dal sistema operativo. Tutti i dispositivi collegati alla porta parallela, ad uno dei vari tipi di bus presenti nel sistema (EIDE, PCI, SCSI, USB), oppure ad una interfaccia PCMCIA richiedono un apposito I/O `driver`.

I compiti dell'I/O `driver` sono essenzialmente quattro:

- avviare l’operazione di I/O offrendo una interfaccia semplificata che nasconda il più possibile le caratteristiche hardware interne del dispositivo;
- forzare la terminazione dell’operazione di I/O se essa non è terminata entro un tempo prefissato; ciò viene realizzato tramite un meccanismo di time-out che invia al NUCLEO un apposito segnale di interruzione quando l’intervallo di tempo è scaduto;
- gestire i segnali di interruzione emessi dal dispositivo risvegliando il processo bloccato in attesa della terminazione dell’operazione di I/O;
- analizzare l’esito dell’operazione di I/O inviando eventuali messaggi di errore nel caso in cui essa non sia andata a buon fine.
11.5 SINCRONIZZAZIONE TRA CPU E DISPOSITIVO DI I/O

Poiché ogni dispositivo di I/O ha caratteristiche peculiari e condizioni di errore proprie (ad esempio, un gestore di stampante deve essere in grado di gestire segnali di fine carta nel contenitore della stampante!), i moderni NUCLEI includono centinaia di gestori distinti.

Allo stesso tempo, l’incessante innovazione tecnologica fa sì che appaiano ogni giorno sul mercato nuovi dispositivi di I/O che devono essere integrati nei NUCLEI esistenti (un esempio è costituito dai lettori DVD apparsi qualche anno fa sul mercato dei personal computer).

Per quanto detto prima, un gestore di dispositivo di I/O non può essere realizzato come un processo di sistema poiché deve essere in grado di gestire un apposito segnale di interruzione; esso deve essere invece integrato nel NUCLEO. In conseguenza, il NUCLEO deve essere predisposto ad accogliere nuovi gestori di dispositivi.

11.5 Sincronizzazione tra CPU e dispositivo di I/O

Non è possibile in questo breve corso dare ulteriori dettagli sulla struttura interna dei driver di I/O. Vale la pena tuttavia accennare al fatto che la sincronizzazione tra CPU e dispositivo di I/O può essere svolta in due modi diversi:

- **Polling**: la CPU interroga periodicamente la porta di I/O per verificare se l’operazione di I/O è terminata. Il ciclo di polling prevede solitamente il rilascio della CPU (una funzione del NUCLEO simile a quella usata per realizzare la chiamata di sistema `sched_yield()`) in modo da consentire ad altri processi di usare la CPU.

- **Interruzioni**: il driver di I/O avvia l’operazione e pone il processo nello stato di blocco sull’evento “interruzione proveniente dal dispositivo”. Il processo verrà posto successivamente nello stato di pronto non appena si sarà verificata l’interruzione richiesta. Quando ciò avviene, il driver di I/O riprende l’esecuzione ed analizza il codice di terminazione dell’operazione di I/O prima di ridare il controllo al processo interessato.

Il primo approccio è preferibile al secondo quando il tempo di risposta del dispositivo di I/O è breve rispetto al tempo richiesto per effettuare la commutazione di processi: le attuali stampanti dotate di buffer di notevoli dimensioni sono solitamente gestite con la tecnica del polling.
Il secondo approccio è usato quando vi sono molti byte da trasferire: i driver di disco fanno uso del DMAC e di interruzioni per segnalare alla CPU la fine dell’operazione di I/O.

11.6 Uso di software cache nei driver per dischi

La caratteristica più importante di un sistema operativo è l’efficienza ed, in particolare, l’efficienza del NUCLEO. Diverse categorie di utenti sottomettono vari sistemi operativi commerciali ad una serie di test impietosi basati sull’uso di programmi di prova (i cosiddetti programmi benchmark) e stabiliscono classifiche non sempre imparziali. Dato che le interfacce verso l’utente risultano oggi abbastanza simili tra loro e che le tecniche di compilazione hanno raggiunto un livello difficilmente migliorabile, l’area in cui si può sperare di raggiungere migliori prestazioni è quella del NUCLEO.

In effetti, i moderni calcolatori spendono oltre il 50% del tempo nello stato Kernel, ossia eseguendo programmi del NUCLEO, per cui risulta cruciale tentare di migliorare le prestazioni di tale componente del sistema operativo.

Ciò premesso, è ragionevole affermare che il collo di bottiglia degli attuali calcolatori è costituito dalla lentezza dei dispositivi di I/O, ed in particolare dei dischi rispetto alle capacità di elaborazione della CPU. Mentre un disco è in grado di trasferire poche centinaia di byte in alcuni millisecondi, la CPU è in grado di eseguire decine di migliaia di istruzioni nello stesso intervallo di tempo!

La risposta dei progettisti di NUCLEI è stata di dedicare una cospicua parte della RAM libera per realizzare una particolare software cache chiamata page cache. Tale cache, da non confondersi con le cache hardware collocate all’interno della CPU, contiene una piccola parte dei dati contenuti nel disco, ossia i dati indirizzati più recentemente. Tipicamente, per un disco avente una capacità di un centinaio Gigabyte si fa uso di una cache di qualche centinaio Megabyte, ossia di una zona di memoria avente una dimensione di due ordini di grandezza inferiore a quella del disco.

Possiamo ricavare la dimensione della page cache sul nostro sistema lanciando il comando `cat /proc/meinfo`. Le prime righe del listato ci forniscono una informazione del tipo:

daniele@alan2:~$ cat /proc/meminfo
MemTotal: 514868 kB
MemFree: 139772 kB
in questo caso, la page cache occupa circa 220 MB su un totale di 512 MB.

Quando viene richiesta una lettura o scrittura sul disco, il NUCLEO verifica se l’informazione richiesta non sia già disponibile nella page cache. Nel caso affermativo, la preleva dalla cache evitando di effettuare un indirizzamento del disco, e quindi con un notevole risparmio di tempo. Se gli indirizzamenti ai blocchi del disco fossero distribuiti in modo uniforme, le page cache non avrebbero ragione di esistere poiché la probabilità di trovare il dato richiesto nella cache sarebbe dell’ordine di uno su mille.

Fortunatamente, la maggior parte dei programmi utenti effettua indirizzamenti al disco in modo “locale”, per cui è plausibile che i programmi lanciati dagli utenti continuino ad operare per qualche tempo sugli stessi blocchi di dati prima di richiederne altri. Per fare un esempio, l’utente effettua prima l’editing del programma sorgente e poi passa a richiedere la compilazione del file editato in precedenza.

L’idea vincente che ha portato all’introduzione della page cache è quella di conservare nella RAM blocchi di dati anche quando nessun processo sta attualmente operando su tali dati. Supponiamo che un processo abbia aperto un file ed abbia quindi effettuato scritture sul file. In questo caso, i blocchi di dati modificati verranno conservati nella cache anche dopo che il processo ha chiuso il file. Altri benefici derivanti dall’uso della page cache riguardano la possibilità di effettuare letture anticipate di file, il cosiddetto read ahead e quella di effettuare scritture differite: un processo di sistema riscrive su disco tutte le pagine “dirty”, ossia le pagine modificata in RAM dai programmi. La scrittura differita consente di sfruttare meglio il disco accorpando scritture su settori contigui di disco in un’unica operazione di I/O.

L’utente può verificare con mano l’effetto benefico sulle prestazioni della page cache ri-eseguendo due volte lo stesso comando e misurandone la durata. Eseguendo ad esempio:

```bash
    time ls -R /usr/src/linux
```

due volte di seguito, si noterà come la seconda esecuzione sia molto più rapida della prima per via dei dati ormai presenti nella disk cache.
Tra i vari compiti del NUCLEO si aggiunge quindi quello di gestire le cache dei dischi e di stabilire una strategia di svuotamento automatico della cache.

In effetti, ogni volta che un settore di disco non si trova nella cache, esso viene letto da disco ed inserito nella cache. Per evitare il riempimento totale della cache, il NUCLEO deve provvedere a riutilizzare alcune parti della cache; a tale scopo, rilascia blocchi di dati nel seguente ordine:

1. quelli letti da disco ma non modificati; tra quelli sceglie quelli non indirizzati da più tempo;
2. se non è stata rilasciata sufficiente memoria dalla cache, quelli letti e modificati; in questo caso, è necessario scrivere su disco la versione aggiornata prima di riutilizzare l’area di memoria.

È interessante osservare che l’uso di cache migliora a tale punto le prestazioni del disco che i dischi più moderni includono spesso una propria cache inclusa nell’unità di controllo del disco che viene gestita interamente in hardware.

Le notevoli prestazioni di tali dischi vanno lette tenendo presente che i tempi medi d’accesso dichiarati non si riferiscono solo alla parte meccanica ma tengono presente anche i benefici indotti dalla cache locale.