Sistemi operativi

Simone Migliazza

Contenuti

Evoluzione

Il compito di un sistema operativo è fornire ai programmi utente un'interfaccia semplificata con l'hardware. La maggior parte dei sistemi operativi reali sono scritti in linguaggio C.

Figure 1.1: Struttura di un generico calcolatore

I dispositivi di ingresso/uscita sono controllati mediante il caricamento di valori in speciali **registri di dispositivo**: per esempio, è possibile ordinare ad un disco di eseguire una lettura caricando nei suoi registri di dispositivo il valore dell'indirizzo su disco, l'indirizzo in memoria principale, numero di byte e direzione (lettura o scrittura).

Il sistema operativo nasconde parzialmente l'hardware e dà al programmatore un insieme di istruzioni più pratico con cui lavorare.

Esso è (normalmente) quella porzione di software che viene eseguita in modalità kernel, i compilatori e gli editor vengono eseguiti in modalità utente. In qualche sistema è difficile tracciare un confine netto: tutto ciò che vie- ne eseguito in modalità kernel fa ovviamente parte del sistema operativo, ma e possibile che alcuni programmi che non vengono eseguiti in questa modalità siano parte di esso o, almeno, gli siano strettamente collegati.

Il sistema operativo è normalmente protetto con meccanismi hardware da ogni tentativo di modifica da parte dell'utente.

1.1 Cos'è un sistema operativo

Il sistema operativo realizza due funzionalita che sono praticamente scorrelate, *estende la macchina* e *gestisce risorse*.

Per quanto riguarda la prima funzionalità: il sistema operativo fornisce una varietà di servizi, di cui i programmi possono usufruire attraverso istruzioni speciali dette chiamate di sistema (**system call**).

E per la seconda: tiene traccia di chi stia usando quale risorsa, soddisfa le richieste delle risorse, contabilizza l'uso e media richieste conflittuali provenienti da programmi e utenti diversi. La gestione delle risorse comporta la loro condivisione sotto due aspetti: rispetto al tempo e rispetto allo spazio. Con quest'ultimo si intende per esempio la condivisione della memoria principale: essa è normalmente ripartita fra diversi programmi in esecuzione, così che possano stare in memoria contemporaneamente (per esempio, allo scopo di fare a turno per l'uso della CPU).

1.2 Storia dei sistemi operativi

1.2.1 La prima generazione

Meta degli anni '40. Macchine enormi, riempivano intere stanze ed erano composte da centinaia di migliaia di **valvole**.

Tutta la programmazione era effettuata in **linguaggio macchina** e non vi era traccia di alcun sistema operativo.

All'inizio degli anni '50 iniziarono ad essere introdotte le **schede perforate**. Ora era possibile scrivere i programmi su di esse e leggerli tramite il calcolatore invece di usare gli spinotti.

1.2.2 La seconda generazione

Innovazione: **transistor**.

Il costo dei calcolatori, di molti milioni di dollari, poteva essere affrontato solo da grosse compagnie, dalle agenzie governative o dalle università. Per far girare un **job** (cioè un programma o un insieme di programmi), un programmatore doveva prima scrivere il programma su carta (in FORTRAN[1] o in assembler), poi doveva copiarlo su schede perforate, infine doveva portare il pacchetto delle schede nella stanza dove venivano raccolti i programmi e darlo ad uno degli operatori. Dopodiché poteva andare a prendersi un caffè in attesa dei risultati in uscita. Appena il calcolatore aveva terminato il job in esecuzione in quel momento, qualsiasi cosa fosse, un operatore andava alla stampante, prelevava quanto prodotto in uscita dal job e lo portava nella stanza dove venivano distribuiti i risultati delle elaborazioni, in modo che il programmatore lo potesse ritirare più tardi; quindi prendeva uno dei pacchetti di schede dalla stanza di raccolta dei programmi e lo faceva leggere al calcolatore. Se risultava necessario anche il compilatore FORTRAN, l'operatore lo doveva prendere dall'archivio e farlo leggere al calcolatore. Gran parte del tempo di elaborazione andava quindi sprecato mentre gli operatori andavano avanti e indietro per la sala macchine.

Per ridurre il tempo la soluzione che venne adottata nella maggior parte dei casi fu quella dei **sistemi batch** (sistemi di elaborazione a lotti). L'idea era quella di raccogliere un intero cassetto di job nella stanza dedicata alla raccolta dei programmi e di farli trasferire su nastro magnetico da un piccolo calcolatore (relativamente) poco costoso, Dopo circa un'ora durante la quale veniva raccolto un batch (lotto) di lavori, il nastro veniva riavvolto e portato nella sala macchine, dove veniva montato sul lettore di nastri. L'operatore caricava quindi un programma speciale (l'antenato degli attuali sistemi operativi) che leggeva il primo job e lo eseguiva; i dati in uscita, anziché venire stampati, venivano registrati su un secondo nastro magnetico. Dopo la fine di ognuno dei job, il sistema operativo leggeva in maniera automatica il prossimo job dal nastro di ingresso e lo mandava in esecuzione. Quando l'intero lotto di lavori era stato eseguito, l'operatore rimuoveva i nastri di ingresso e di uscita, li sostituiva con quelli di un nuovo lotto di lavori e prendeva il nastro di uscita per stamparne i dati fuori linea, cioè senza essere direttamente collegato al calcolatore principale.

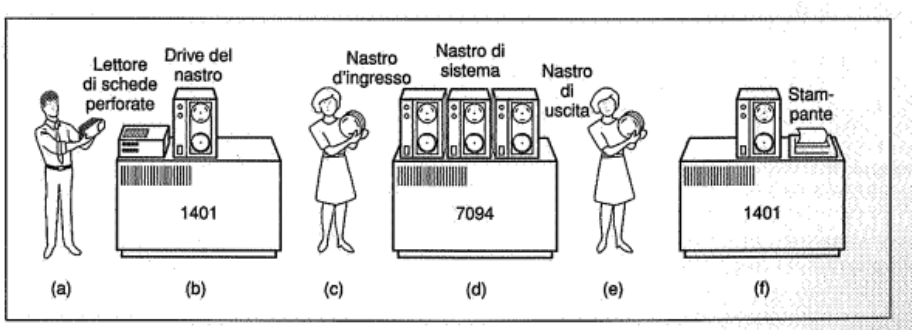


Figure 1.2: Uno dei primi sistemi di elaborazione batch

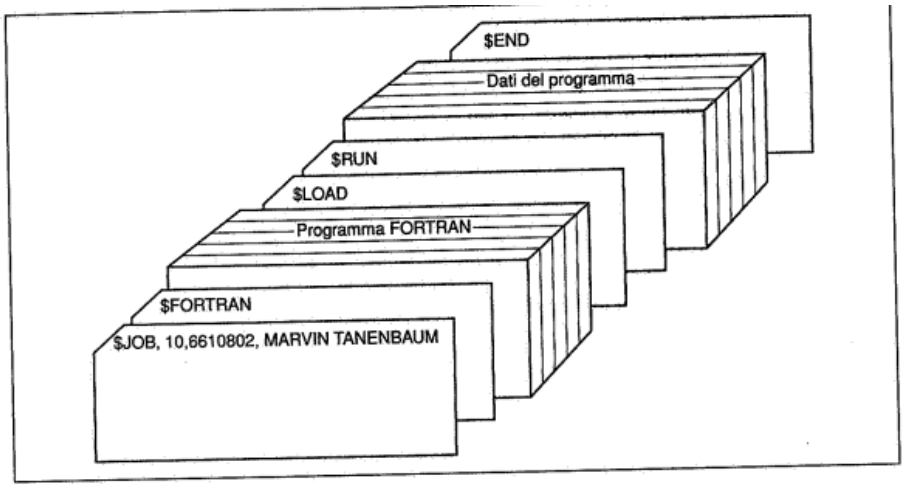


Figure 1.3: Struttura di un tipico job FMS

[1] *FOR*mule *TRAN*slator

1.2.3 La terza generazione

Innovazione: **circuiti integrati**.

I sistemi di terza generazione resero popolare la tecnica della **multiprogrammazione**: essa consisteva nel dividere la memoria centrale in alcune partizioni, con un job diverso per ogni partizione; mentre un job rimaneva in attesa del completamento di una certa attività di ingresso/uscita, la CPU poteva essere usata da un altro job e quando si riusciva a tenere in memoria un numero sufficiente di job contemporaneamente, si riusciva a mantenere occupata la CPU quasi per il 100 per cento del tempo.

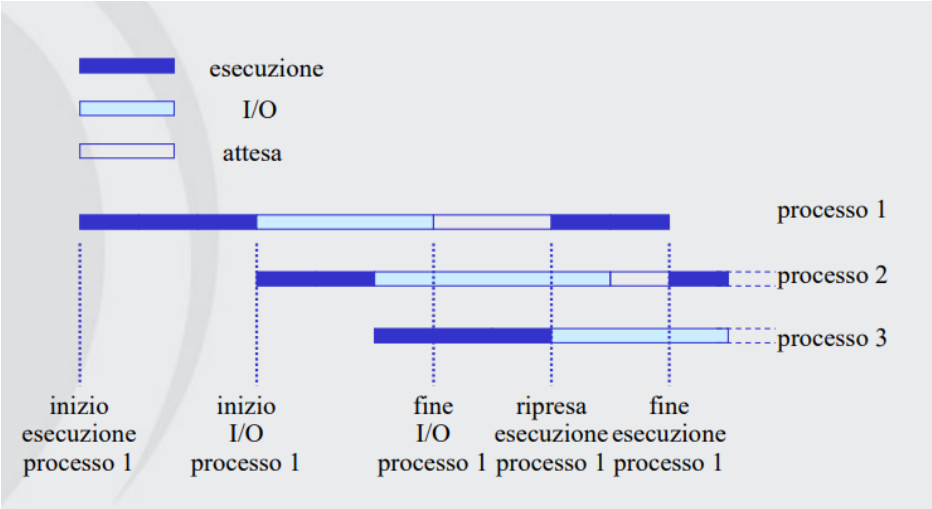


Figure 1.4: Esempio di multitasking

Un'altra caratteristica importante dei sistemi operativi della terza generazione era quella di permettere la lettura dei job dalle schede al disco non appena le schede venivano portate alla sala macchine. Poi, non appena il job in esecuzione terminava, il sistema operativo poteva caricare un nuovo job dal disco nella partizione di memoria che si era resa disponibile e mandarlo in esecuzione. Questa tecnica viene chiamata **spooling**.

Il desiderio di un tempo di risposta veloce preparò la strada al **timesharing** (condivisione di tempo), una variante della multiprogrammazione nella quale ogni utente aveva a disposizione un terminale in linea. Nei sistemi timesharing, se 20 utenti erano collegati contemporaneamente e 17 di loro stavano pensando, bevendo caffè o parlando, la CPU poteva essere assegnata all'esecuzione a turno dei tre job che in quel momento richiedevano il servizio.

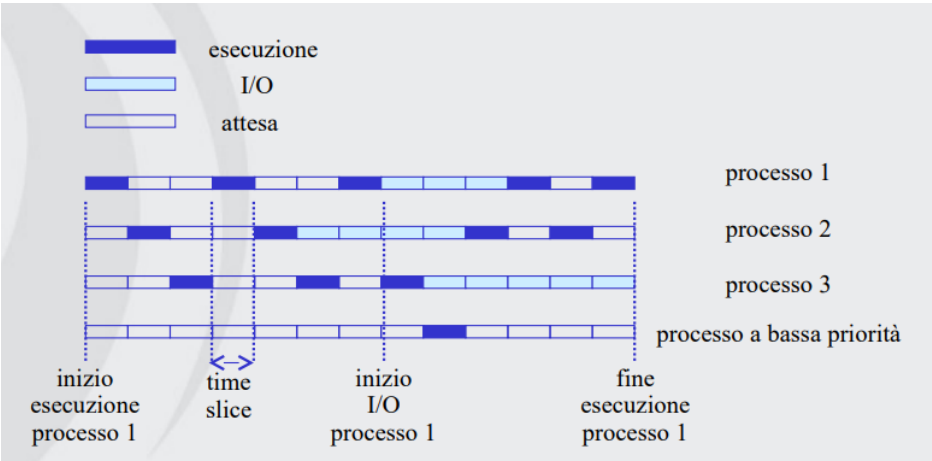


Figure 1.5: Esempio di condivisione del tempo

1.2.4 La quarta generazione

Innovazione: **VLSI** (Very Large Scale Integration).

Nasce l'era dei personal computer. MS-DOS domina il mercato.

Negli anni ’60 viene introdotta per la prima volta la **GUI**, completa di finestre, icone, menù e mouse.

Negli anni ’80 iniziano ad essere utilizzati i **sistemi operativi di rete** e i **sistemi operativi distribuiti**.

Nei primi gli utenti sono a conoscenza dell’elevato numero di calcolatori e possono accedere alle macchine remote; ogni macchina ha il proprio sistema operativo e i propri utenti.

Nei secondi invece, il sistema distribuito appare agli utenti come un singolo sistema monoprocessore. Essi non sanno dove i propri file sono allocati e non sanno dove i loro programmi vengono fatti girare.

1.2.5 La quinta generazione

Sono i sistemi operativi mobile.

Tra la grande varietà di sistemi operativi presenti sul mercato attuale si vuole dare una spiccata importanza ai sistemi operativi **real-time**. Quando deve essere eseguita un azione *assolutamente* in un dato momento o intervallo di tempo si parla di sistemi real-time. Una categoria esemplificativa sono i sistemi digitali audio o quelli multimediali, essi fanno parte della categoria *soft real-time systems*.

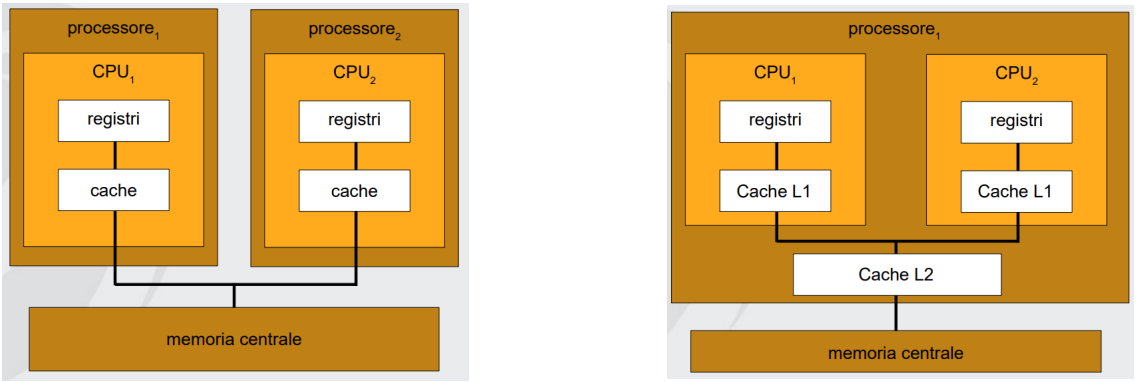
1.3 L’hardware dei calcolatori

I componenti principali che compongono un calcolatore sono:

- Processore
- Memoria
- Dispositivi in/out
- Bus

1.3.1 Processore

Il processore è un chip che contiene una o più CPU. Il ”cervello” di un calcolatore è la CPU, che recupera le istruzioni dalla memoria e le esegue. Essa piò essere composta da più *core*, che sono l’unità di elaborazione base di una CPU.



(a) multiprocessore

(b) multicore

Figure 1.6: Architetture con multieleborazione

Occorre fare attenzione poiché aggiungere nuove CPU aumenta le prestazioni relative al calcolo ma peggiora quelle relative all’accesso alla memoria. Una possibile soluzione tuttavia può essere fornire a ciascuna CPU una propria memoria locale accessibile da un piccolo bus. I sistemi che implementano questa soluzione sono detti sistemi **NUMA**: ovvero ad accesso non uniforme alla memoria. La pecca più grande è lo spreco di tempo in quei casi in cui una CPU deve accedere alla memoria di un’altra.

Il ciclo di azioni base eseguito da ogni CPU consiste nel recuperare la prima istruzione dalla memoria, decodificarla per determinarne tipo e operandi, ed eseguirla e quindi recuperare, decodificare, eseguire le istruzioni seguenti. In questo modo vengono completati i programmi. Ogni CPU è in grado di eseguire un insieme di istruzioni specifico.

Tutte le CPU contengono, al loro interno, alcuni registri per mantenervi variabili chiave e risultati temporanei. Normalmente, l’insieme di istruzioni ne contiene quindi alcune per caricare una parola dalla memoria in un registro e per depositare una parola da un registro alla memoria.

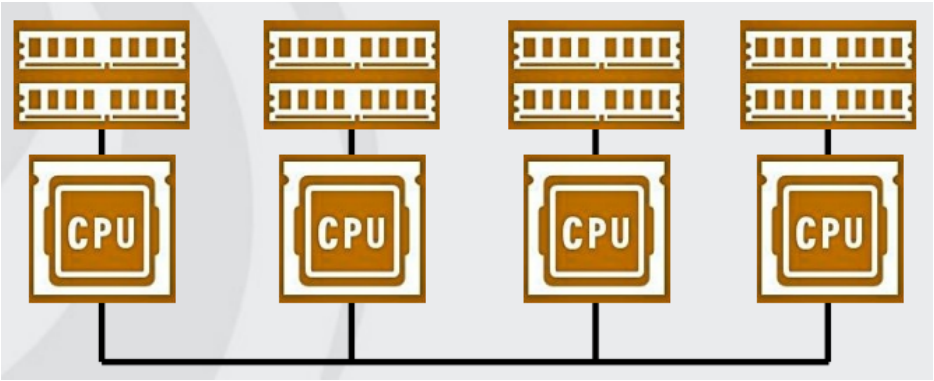


Figure 1.7: sistema NUMA

La maggior parte dei calcolatori ha parecchi registri speciali disponibili al programmatore. Uno di questi è il *program counter*, o contatore di programma, che contiene l'indirizzo di memoria della prossima istruzione da andare a prendere (*fetch*); dopo che l'istruzione è stata recuperata, il program counter viene aggiornato per puntare all'istruzione successiva. Un altro registro è lo *stack pointer* o puntatore alla pila, che punta alla cima dello stack corrente in memoria; tale stack o pila contiene una struttura (frame) per ogni procedura che è stata iniziata ma non è ancora terminata, ognuna delle quali contiene i parametri d'ingresso, e le variabili locali e temporanee che non vengono mantenuti nei registri.

Le CPU moderne hanno unità separate per le tre fasi di recupero, decodifica e esecuzione, così che, mentre è in esecuzione l'istruzione n, potrebbero essere in grado di decodificare l'istruzione n+1 e recuperare l'istruzione n+2. Un'organizzazione di questo tipo è detta *pipeline*.

La maggior parte delle CPU hanno due modalità, quella kernel e quella utente. Il sistema operativo viene eseguito nella prima modalità mentre i programmi utente nella seconda. Per ottenere servizi dal sistema operativo, un programma deve eseguire una chiamata di sistema o system call, che esegue una trap al kernel e invoca il sistema operativo: l'istruzione TRAP cambia da modalità utente a modalità kernel e avvia il sistema operativo. Gli elaboratori hanno altre istruzioni di tipo trap oltre a quella per eseguire chiamate di sistema: la maggior parte delle trap sono causate dall'hardware per avvisare di una situazione eccezionale, come può essere un tentativo di divisione per 0 o un underflow nell'aritmetica a virgola mobile. In ogni caso, il sistema operativo ottiene il controllo, e deve decidere cosa fare.

1.3.2 Memoria (facoltativo)

La memoria ideale dovrebbe essere estremamente veloce (più veloce dell'esecuzione di un'istruzione, in modo che la CPU non sia rallentata). Essa è costruita come una gerarchia di strati.

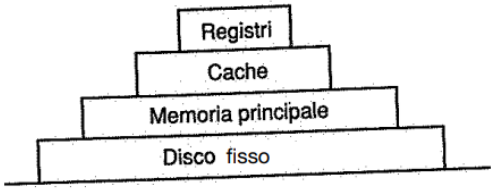


Figure 1.8: Tipica gerarchia di una memoria

- i *registri*: interni alla CPU.
- la *cache*: controllata dall'hardware.
- la *memoria principale*: anche chiamata **RAM** (Random Access Memory).
- *hard disk*.

In aggiunta ai tipi di memoria discussi sopra, molti elaboratori hanno una piccola porzione di memoria ad accesso casuale non volatile; a differenza della RAM, la memoria non volatile non perde il suo contenuto nel momento in cui viene tolta la corrente. La **ROM** (Read Only Memory) è programmata in fabbrica e non può essere modificata in seguito. Su qualche elaboratore, il programma di avvio (bootstrap loader) usato per avviare l'elaboratore stesso è contenuto in ROM.

Anche le EEPROM (Electrically Erasable ROM, ROM cancellabili elettricamente) e le flash RAM non sono volatili, ma al contrario delle ROM possono essere cancellate e riscritte. Tuttavia, per scrivere su di esse occorrono tempi di qualche ordine di grandezza più lunghi di quelli per scrivere sulle RAM,

Un altro tipo di memoria è la memoria CMOS, che è volatile ed è utilizzata da molti elaboratori per memorizzare ora e data corrente. La memoria CMOS è utilizzata perché consuma così poca corrente che la batteria originale

installata in fabbrica dura spesso diversi anni. Tuttavia, quando inizia a scaricarsi, può sembrare che l'elaboratore abbia il morbo di Alzheimer, dimenticandosi cose che ha saputo per anni, come il disco dal quale eseguire il boot.

Per tenere due o più programmi in memoria principale occorre risolvere due problemi:

1. gestire la rilocazione.
2. proteggere i programmi l'uno dall'altro e il kernel dai programmi.

Il primo si risolve nella conversione dell'indirizzo generato dal programma, chiamato indirizzo virtuale, nell'indirizzo utilizzato dalla memoria, chiamato indirizzo fisico. Il dispositivo che effettua il controllo e la corrispondenza è detto MMU (Memory Management Unit). Per il secondo si usa una coppia di registri speciali, il *registro base* e il *registro limite*. Quando un programma viene eseguito, il registro base è impostato al punto di inizio del suo codice,e il registro limite contiene il totale dello spazio occupato dal codice e dai dati del programma. Quando un'istruzione deve essere recuperata, l'hardware controlla che il program counter sia minore del registro limite e, se è così, lo somma al contenuto del registro base e spedisce il risultato alla memoria.Il registro base rende impossibile che il programma si riferisca a parti di memoria al di sotto di se stesso, e il registro limite rende impossibile che si riferisca a parti di memoria superiori al programma stesso.

Il passare da un programma ad un altro è generalmente una operazione costosa, essa è denominata *cambio di contesto*.

1.3.3 I dispositivi input/output (facoltativo)

I dispositivi di ingresso/uscita generalmente sono composti da due parti: un controllore ed il dispositivo stesso.

Il controllore è un chip posto su una scheda estraibile, che controlla fisicamente il dispositivo, ed accetta comandi dal sistema operativo. Il suo compito è quello di presentare un interfaccia semplificata al sistema operativo. I controllori spesso contengono piccoli elaboratori embedded.

Dato che ogni tipo di controllore è diverso, sono necessari software diversi per controllare ognuno di essi; il software che parla al controllore, che da i comandi e accetta le risposte, è detto device *driver*. Ogni fabbricante di controllori deve fornire un driver per ogni sistema operativo che supporta. Per essere usato, un driver deve essere posto nel sistema operativo per essere eseguito in modalità kernel: occorre quindi un riavvio ogni volta che va installato un nuovo driver (tranne in particolari casi come per gli *hot pluggable device*, in quel caso si fa in modo che non sia necessario). Ogni controllore ha un piccolo numero di registri che vengono usati per comunicare con esso. Per esempio, un controllore di disco minimale potrebbe avere registri che specificano l'indirizzo su disco, l'indirizzo di memoria, il numero di settori e la direzione (lettura o scrittura); per attivare il controllore, il driver riceve un comando dal sistema operativo, quindi lo traduce nei valori appropriati da scrivere nei registri del dispositivo. Su qualche elaboratore, i registri di dispositivo sono messi in corrispondenza con indirizzi all'interno dello spazio di indirizzamento del sistema operativo.

Ci sono tre modi per eseguire operazioni di ingresso/uscita:

1. Polling.
2. Interrupt.
3. Accesso diretto alla memoria (*DMA*).

1.3.4 I bus (facoltativo)

L'USB (Universal Serial Bus) è stato inventato per permettere di collegare al calcolatore tutti i dispositivi di ingresso/uscita lenti, come la tastiera e il mouse. Utilizza un piccolo connettore a quattro fili, due dei quali forniscono energia elettrica ai dispositivi USB. Tutti i dispositivi USB condividono un solo driver di dispositivo USB.

Il brutto arrivava quando l'utente comprava una scheda sonora e un modem e capitava che entrambi usassero, diciamo, l'interruzione 4. Per ovviare al problema si utilizza il *plug and play*, esso permette al sistema di raccogliere automaticamente le informazioni relative ai dispositivi di ingresso/uscita, assegnare in modo centralizzato i livelli di interruzione e gli indirizzi di ingresso/uscita, e comunicare ad ogni scheda il proprio numero.

Sulla scheda madre risiede un programma detto *BIOS* (Basic Input Output System), che contiene software di basso livello per l'ingresso/uscita, comprese le procedure per leggere la tastiera, scrivere sul video e fare ingresso/uscita su disco. Oggigiorno, è contenuto in una flash RAM, che è non volatile e può essere aggiornata dal sistema operativo quando vengono rilevati dei bachi nel BIOS. Il BIOS determina il dispositivo di memoria da cui effettuare l'inizializzazione (boot) provando i dispositivi elencati in una lista memorizzata nella memoria CMOS, che l'utente può modificare eseguendo il configuratore del BIOS immediatamente dopo aver acceso il calcolatore.

Struttura di un sistema operativo

2.1 I componenti del sistema

2.1.1 I processi

Un **processo** è essenzialmente un programma in esecuzione. Ad ogni processo vengono associati il proprio **spazio di indirizzamento** e una lista di locazioni in memoria nella quale il processo può leggere e scrivere.

Lo spazio di indirizzamento (chiamato anche *immagine in memoria* del processo) contiene il programma stesso, i suoi dati e il suo stack. Inoltre, ad ogni processo viene associato un insieme di registri, tra cui il sio program counter, lo stack pointer e altri...

Il modo più semplice per avere una buona idea intuitiva di processo è di pensare ai sistemi timesharing. Periodicamente il sistema operativo decide di sospendere l'esecuzione di un processo e di iniziare ad eseguirne un altro perché, ad esempio, il primo ha esaurito il tempo di CPU che aveva a disposizione. Tutte le informazioni relative al processo in fase di sospensione devono essere salvate. Le informazioni (tranne il contenuto del suo spazio di indirizzamento) vengono salvate in una tabella di sistema operativo chiamata **tabella dei processi**, che essenzialmente è un array di strutture.

Le principali chiamate di sistema per la gestione dei processi sono quelle che si occupano della creazione e della terminazione dei processi. Un processo può creare uno o più processi, detti processi **figli**. La *comunicazione tra processi* è un argomento importante che riprenderemo in seguito. Ci sono altre chiamate di sistema utilizzate, per esempio: richiedere più memoria; liberare memoria inutilizzata; attendere la terminazione di un processo figlio; sovrapporre un altro programma al proprio.

Talvolta è necessario inviare informazioni ad un processo che non le sta aspettando: una volta inviato il messaggio si fa partire un timer, nel caso in cui non arrivi un segnale di acknowledgement allo scadere del timer il sistema operativo manda un segnale di allarme che permette al processo di interrompere la attuale attività per gestire la situazione. Questi segnali sono l'equivalente software delle *interrupt* hardware. Anche molte delle interruzioni generate da situazioni irregolari (*trap*) rivelate dall'hardware, come un tentativo di divisione per zero per esempio, sono convertite in segnali indirizzati al processo, che gestirà la situazione.

2.1.2 La gestione della memoria

I sistemi operativi permettono di mantenere più programmi in memoria contemporaneamente. Per evitare che interferiscano fra loro, o con il sistema operativo, è necessario un meccanismo di protezione hardware, controllabile da sistema operativo. Cosa accade se un processo ha uno spazio di indirizzamento più grande della memoria principale e lo vuole usare tutto? si usa la tecnica della **memoria virtuale**: essa prevede di mantenere alcuni dati su disco e spostare i vari frammenti di dati, all'occorrenza, dal disco alla memoria principale.

2.1.3 L'ingresso/uscita

Ogni sistema operativo ha un sottosistema per gestire i dispositivi di ingresso/uscita. Una parte di quel software è indipendente dal dispositivo mentre un'altra no (il driver del dispositivo).

2.1.4 I file

Un concetto supportato da tutti i sistemi operativi è il **file system**. Esso ne è una delle funzioni principali. Il sistema deve presentare un modello pulito ed astratto di file, indipendente dai dispositivi di immagazzinamento. Le chiamate di sistema sono necessarie per rimuovere, leggere o scrivere file. La maggior parte dei sistemi operativi supporta il concetto di directory, il posto dove tenere i file.

Ciascun file all'interno della gerarchia può essere specificato indicandone il *path name* a partire dalla cima della gerarchia (*root directory*). Il cammino si dice *assoluto* se parte dalla root e *relativo* se parte da *directory di lavoro*.

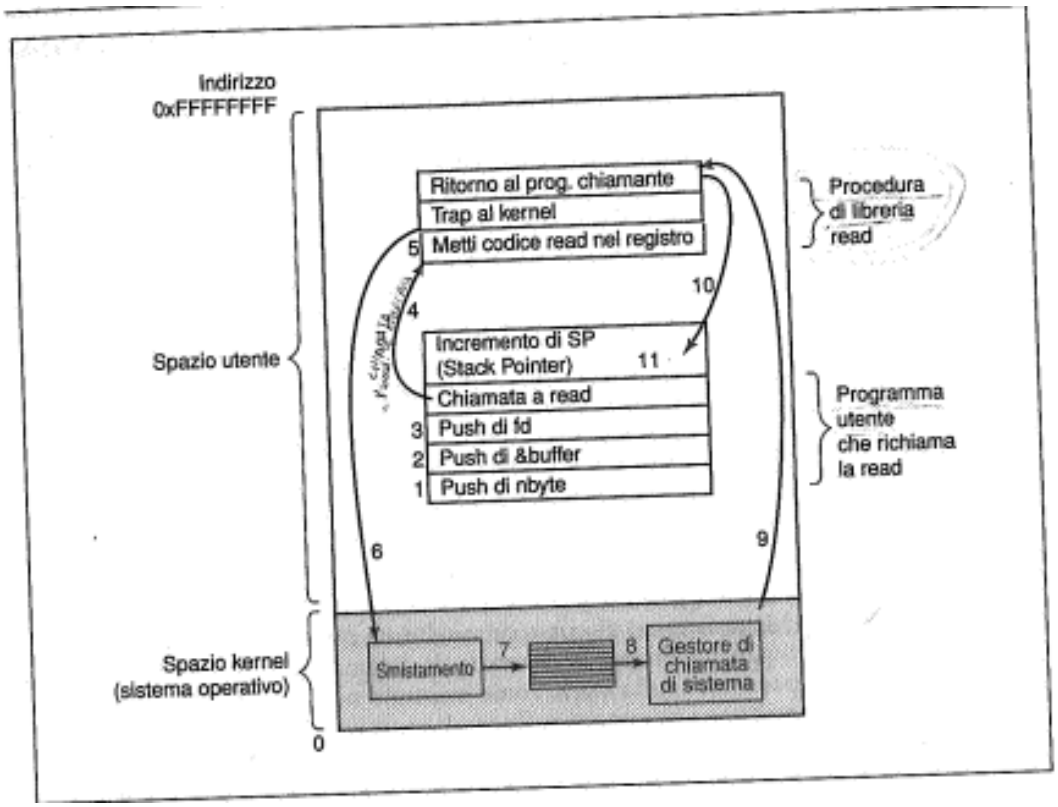


Figure 2.1: Gli 11 passi della chiamata di sistema **read(fd, buffer, nbyte)**

In UNIX un concetto importante è quello di file system *montato*: è possibile attaccare un file system di un disco ad un altro montando la root di uno in una directory dell'altro, occorre fare attenzione a tenere la directory in cui si monta vuota poiché i file all'interno saranno inaccessibili finché rimarra montato il secondo disco. Il comando in questione è **mount**.

Un altro concetto importante in UNIX è quello di *file speciale*: essi sono utilizzati per fare in modo di trattare i dispositivi ingresso/uscita esattamente come file, usando le stesse chiamate di sistema.

Per la comunicazione tra processi si utilizza il concetto di **pipe**: essa è una specie di pseudo-file che collega due processi. Essi scrivono e leggono sulla pipe come se fosse un file.

2.1.5 Sicurezza

I calcolatori contengono informazioni che gli utenti potrebbero voler mantenere nascoste. In UNIX i file sono protetti associando a ciascuno un codice, i bit *rwx*, per esempio: *rwxr-x–x* significa che l'utente proprietario può leggere scrivere ed eseguire il file, i membri che appartengono al suo gruppo non possono scrivere e tutti gli altri possono solo leggere.

2.1.6 La shell

La shell è l'interprete dei comandi UNIX, non è parte del sistema operativo ma ne fa un pesante utilizzo. Essa usa il terminale come ingresso e uscita standard, e parte scrivendo un *prompt*, un carattere come il segno $, che avvisa che la shell è in attesa di comandi.

2.2 I servizi del sistema: le system call

L'interfaccia tra sistema operativo e programmi utente è definita dall'insieme di system call fornite. Esse variano tra i sistemi operativi, qui ci si riferirà allo standard POSIX, da cui UNIX (la maggior parte dei sistemi operativi esegue le stesse funzioni con dettagli diversi).

Per rendere più chiaro il meccanismo viene presentato un esempio: la chiamata di sistema *read*. Essa ha tre parametri: il file da leggere, un buffer e il numero di byte da leggere. Come quasi tutte le chiamate di sistema viene chiamata da programmi C chiamando una procedura di libreria con lo stesso nome della chiamata di sistema:

```
cont = read(file, buffer, nbyte);
```

Se la chiamata di sistema non può essere completata il numero dell'errore viene messo in una variabile globale: *errno*.

POSIX ha circa 100 chiamate di procedura, alcune delle più importanti sono elencate nella figura 2.2. Nota bene: la corrispondenza tra chiamate di procedura POSIX e chiamate di sistema non è detto che sia 1:1.

Gestione dei processi	
Chiamata	Descrizione
pid = fork ()	Crea un processo figlio identico al genitore
pid = waitpid(pid, &statloc, opzioni)	Aspetta che il figlio termini
s = execve(nome, argv, ambp)	Sostituisce l'immagine del processo
exit(stato)	Termina l'esecuzione del processo e restituisce lo stato

Gestione dei file	
Chiamata	Descrizione
fd = open(file, come, ...)	Apre un file in lettura, scrittura o entrambe
s = close(fd)	Chiude un file aperto
n = read(fd, buffer, nbyte)	Legge dati da un file e li mette in un buffer
n = write(fd, buffer, nbyte)	Scrive dati in un file, prendendoli da un buffer
posizione = lseek(fd, offset, da_dove)	Muove il puntatore a file
s = stat(nome, &buf)	Restituisce le informazioni relative allo stato di un file

Gestione delle directory e del file system	
Chiamata	Descrizione
s = mkdir(nome, modo)	Crea una nuova directory
s = rmdir(nome)	Cancella una directory vuota
s = link(nome1, nome2)	Crea un nuovo riferimento, nome2, che punta a nome1
s = unlink(nome)	Cancella un elemento di una directory
s = mount(speciale, nome, flag)	Fa il mount di un file system
s = umount(speciale)	Fa l'umount di un file system

Miscellanea	
Chiamata	Descrizione
s = chdir(nomedir)	Cambia la directory di lavoro
s = chmod(nome, modo)	Cambia i bit di protezione di un file
s = kill(pid, segnale)	Manda un segnale a un processo
secondi = time(&secondi)	Restituisce il tempo trascorso dal 1 gennaio 1970

Figure 2.2: Alcune delle chiamate di sistema POSIX.

2.3 La struttura dei sistemi

Esamineremo le diverse strutture di sistemi operativi che sono state provate:

1. sistemi monolitici
2. sistemi a livelli
3. macchine virtuali
4. sistemi client-server

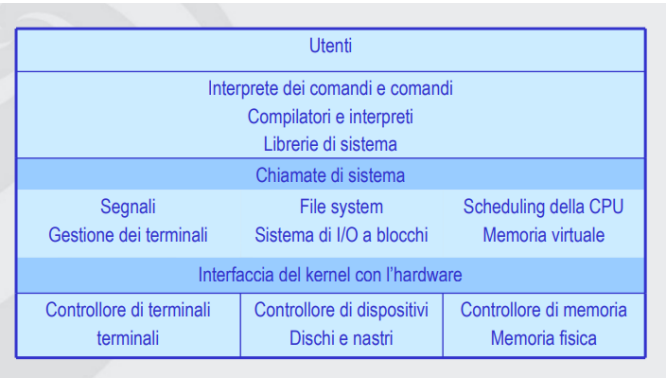
2.3.1 Sistemi monolitici

Il sistema operativo, in questo caso, come un insieme di procedure, ciascuna delle quali può chiamare una qualunque delle altre, quando ne ha bisogno. Per costruire il vero programma oggetto del sistema operativo, prima si compilano le singole procedure, o i file che contengono le procedure, e in seguito vengono fatte legare in un unico file dal *linker* di sistema. Ogni procedura è visibile dalle altre.

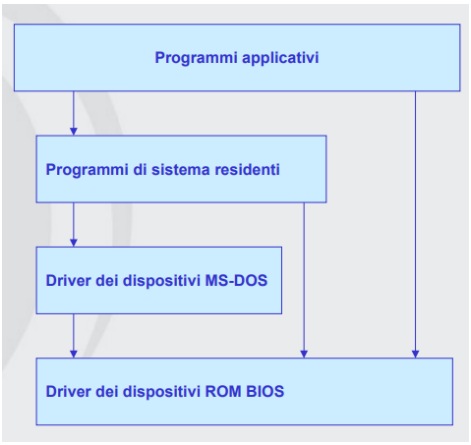
I servizi forniti dal sistema operativo (le system call), vengono richiesti mettendo i parametri in luoghi prefissati (per esempio lo stack) ed in seguito viene eseguita un' istruzione *trap*. Questa istruzione fa passare la macchina da user mode a kernel mode, trasferendo il controllo al sistema operativo. Il sistema operativo controlla i parametri, capisce quale chiamata di sistema eseguire e la esegue.

UNIX	Win32	Descrizione
fork	CreateProcess	Crea un nuovo processo
waitpid	WaitForSingleObject	Aspetta che un processo termini
execve	(nessuna)	CreateProcess = fork + execve
exit	ExitProcess	Termina l'esecuzione
open	CreateFile	Crea un file o apre un file esistente
close	CloseHandle	Chiude un file
read	ReadFile	Legge dati da un file
write	WriteFile	Scrive dati in un file
lseek	SetFilePointer	Muove il file pointer (riferimento alla posizione interna al file)
stat	GetFileAttributesEx	Restituisce vari attributi di file
mkdir	CreateDirectory	Crea una nuova directory
rmdir	RemoveDirectory	Rimuove una directory vuota
link	(nessuna)	Win32 non supporta i link (collegamenti)
unlink	DeleteFile	Elimina un file esistente
mount	(nessuna)	Win32 non supporta il mount
umount	(nessuna)	Win32 non supporta il mount
chdir	SetCurrentDirectory	Cambia la directory di lavoro corrente
chmod	(nessuna)	Win 32 non supporta chiamate per la sicurezza (ma NT sì)
kill	(nessuna)	Win 32 non supporta i segnali
time	GetLocalTime	Restituisce data e ora correnti

Figure 2.3: UNIX vs. Win32 API



(a) Struttura UNIX



(b) Struttura MS-DOS

Figure 2.4: Esempi di strutture monolitiche

Lo schema che si viene a creare è tipo:

1. Un programma principale chiama per le procedure di servizio richieste.
2. Un insieme di procedure di servizio esegue le chiamate di sistema.
3. Un insieme di procedure di utilità aiuta le procedure di servizio.

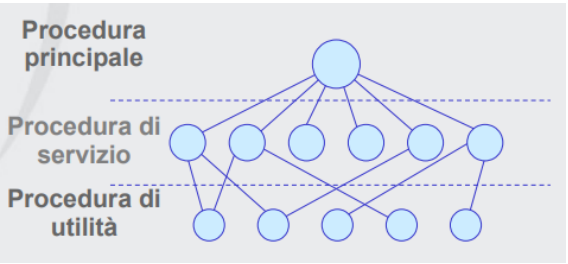


Figure 2.5: Un semplice modello di strutturazione per un sistema monolitico.

2.3.2 Sistemi a livelli

Il primo sistema a livelli è stato il sistema **THE**. Esso aveva 6 livelli:

- livello 0, allocazione del processore e salta da un processo all'altro quando si verificano interrupt o timer scaduti.
- livello 1, gestione della memoria principale.

- livello 2, gestione delle comunicazioni processo-console.
- livello 3, gestione input/output.
- livello 4, programmi utente.
- livello 5, processo dell'operatore.

2.3.3 Macchine virtuali

Un sistema time-sharing fornisce (1) multiprogrammazione e (2) una macchina estesa con un'interfaccia più adeguata all'hardware. L'essenza delle macchine virtuali sta nel separare completamente le due funzioni.

Il cuore del sistema è detto *monitor della macchina virtuale*. Gira direttamente sull'hardware e si occupa della multiprogrammazione, fornendo al livello superiore parecchio macchine virtuali. Queste macchine virtuali sono *copie* esatte del semplice hardware, incluse modalità utente/kernel, in/out, interrupt...

Una particolare area dove vengono usate le macchine virtuali è per eseguire programmi java. Si utilizza infatti la *Java Virtual Machine*.

2.3.4 Il modello client-server

Una tendenza dei moderni sistemi operativi è quella di sviluppare ulteriormente l'idea di spostare il codice verso i livelli superiori e rimuoverne dal sistema operativo, lasciando un cosiddetto *microkernel*. L'approccio più comune implementa la maggior parte delle funzioni del sistema operativo attraverso i processi utente. Un processo utente (**client**), per richiedere la lettura di un blocco di memoria spedisce la richiesta ad un processo **server** che la elabora e restituisce il risultato.

Il kernel si occupa soltanto della comunicazione client server, dividendo il sistema operativo in parti, ciascuna con uno specifico compito.

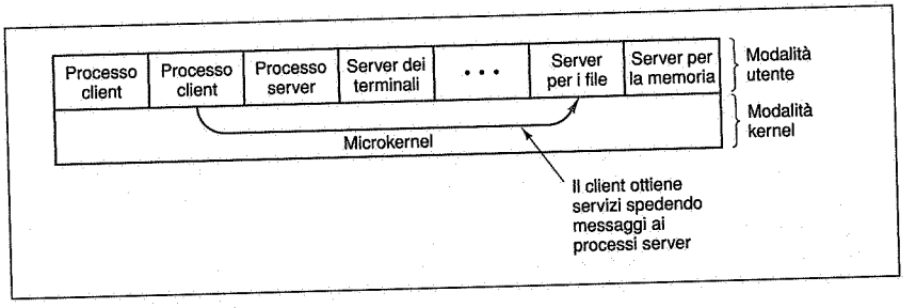


Figure 2.6: Il modello client-server

Un altro vantaggio di questo modello è il suo possibile utilizzo nei sistemi distribuiti. Accade sempre la stessa cosa: richiesta e risposta.

Va sottolineato che la figura 2.7 non è completamente corretta, alcune funzioni non sono realizzabili da programmi che girano nello spazio utente. Due soluzioni:

1. consentire a processi critici (es: driver di dispositivi) di essere eseguiti in modalità kernel.
2. lasciare le politiche di decisione ai processi server che vengono eseguiti nello spazio utente.

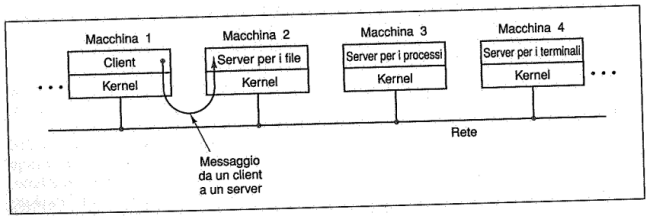


Figure 2.7: Il modello client-server in un sistema distribuito

I processi

Un processo è l'astrazione che indica un programma in esecuzione. In un sistema multiprogrammato la CPU passa da un programma all'altro, eseguendo ciascuno di essi per decine o centinaia di millisecondi; ogni secondo essa può lavorare su tanti programmi diversi senza che l'utente se ne accorga. Per indicare questo continuo alternarsi si usa l'espressione **pseudo-parallelismo**, diverso dal vero parallelismo dei sistemi multiprocessore (con più CPU quindi).

E' comunque impensabile utilizzare il 100% della CPU, tuttavia avendo tanti processi in esecuzione ci si avvicina parecchio, vedi la figura 3.1:

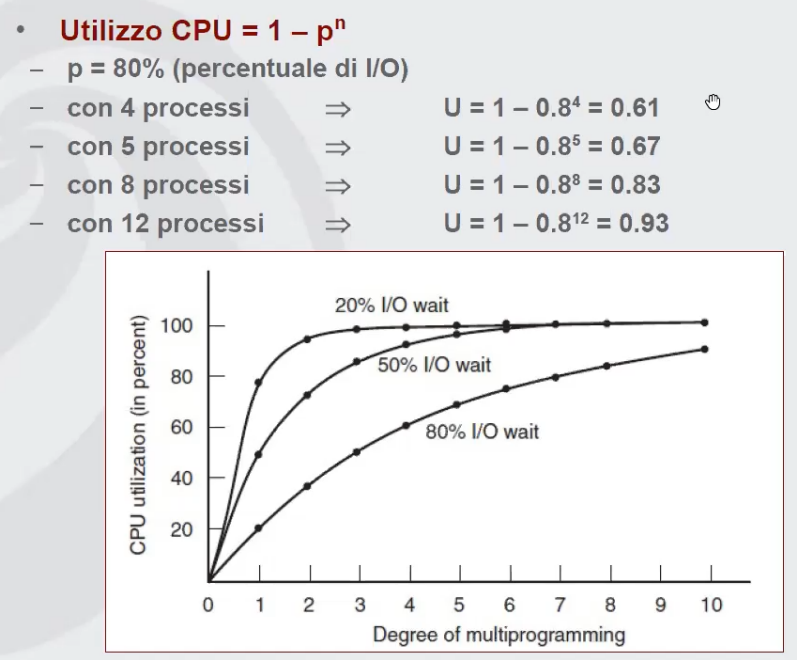


Figure 3.1: stima di utilizzo della cpu

3.1 Il modello a processi

Ogni processo ha il suo flusso di controllo indipendente dagli altri. Nella programmazione di processi **non** si devono fare ipotesi sulla temporizzazione. Quando un processo ha requisiti real-time critici bisogna prendere provvedimenti opportuni in modo tale che i requisiti vengano soddisfatti.

3.2 La creazione dei processi

Nei sistemi occorre avere un meccanismo per creare e distruggere i processi. A provocare la creazione di un processo sono, principalmente, questi eventi:

1. L'inizializzazione del sistema.
2. L'esecuzione di una chiamata di sistema per la creazione di un processo, effettuata da un processo in esecuzione.
3. Una richiesta da parte dell'utente.

4. L'inizio di un *job batch*[1] .

Quando un sistema operativo viene lanciato vengono creati diversi processi, alcuni che interagiscono con l'utente (*foreground process*) e altri che sono eseguiti di nascosto (*background process*). Questi ultimi vengono anche chiamati demoni (**daemon**). In UNIX si usa il programma **ps** per avere informazioni sui processi attivi, in windows si premono *CTRL-ALT-CANC*. Con il comando *ps* viene scattata un istantanea del sistema, per avere informazioni in tempo reale si usa il programma **top**.

In UNIX esiste una sola chiamata di sistema che crea un nuovo processo: **fork**, essa crea una copia esatta del processo chiamante. Dopo la fork i processi padre e figlio sono identici, hanno la stessa immagine di memoria (con diversi spazi di indirizzamento coinvolti, a meno di alcune eccezioni), le stesse stringhe di ambiente...

Normalmente il processo figlio esegue poi una **execve**, o una chiamata simile, per cambiare la sua immagine di memoria e eseguire un nuovo programma. La ragione di questo procedimento in due passi è di permettere al figlio di manipolare i propri descrittori di file, dopo la fork e prima della execve, per realizzare la ridirezione di stdin, stdout e stderr.

3.3 La terminazione dei processi

Un processo termina al verificarsi di una di queste condizioni:

1. Terminazione normale (volontaria).
2. Terminazione con errore (volontaria).
3. Fatal error (involontaria).
4. Kill da un altro processo (involontaria).

La prima si verifica con una chiamata di sistema che avvisa il sistema operativo di aver terminato i propri compiti, in UNIX è la chiamata **exit**. La seconda si verifica al presentarsi di un errore controllato dal programma. La terza invece si presenta all'occorrere di un bug del codice, che causa l'esecuzione di istruzioni illegali (e.g. divisione x zero). Per la quarta va fatto notare che il processo killer deve avere le autorizzazioni necessarie all'uccisione.

3.4 Gerarchie di processi

Il processo figlio può creare altri processi, questo porta alla formazione di una gerarchia di processi (non in windows, lì non c'è una gerarchia). In particolare possiamo vedere come viene inizializzato UNIX: un processo speciale detto **init** è presente nell'immagine di avvio e viene lanciato. Quando inizia la sua esecuzione legge un file che dice quanti terminali ci sono, quindi si biforca creando un nuovo processo per terminale e questi processi aspettano che venga effettuato il login. Una volta effettuato il processo di login si biforca ulteriormente.. e così via. Così tutti i processi hanno come root *init*. Per verificarlo usa il comando **pstree -p**.

Un esempio di programma in C che mostra la creazione di un processo figlio:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    /* crea un duplicato del padre */
    int pid=fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    }
    else if (pid == 0) {
        /* processo figlio */

        /* viene eseguito un nuovo programma */
        execl("/bin/ls", "ls", NULL);
```

[1] Riguarda sistemi batch che si trovano in grossi mainframe, dove gli utenti possono inviare al sistema job batch anche in remoto. Quando il sistema operativo stabilisce di avere abbastanza risorse per eseguire un nuovo job crea un nuovo processo ed esegue il job successivo dalla coda.

```
    }
    else {
        /* processo padre */

        /* attende che il figlio termini */
        wait(NULL);
        printf("Child Complete\n");
        exit(0);
    }
    return 1; /* non dovrebbe accadere */
}
```

3.5 Gli stati dei processi

I processi devono spesso interagire tra loro, per esempio uno potrebbe voler prendere in ingresso i dati di uscita di un altro:

```
cat cap1 cap2 | grep tree
```

In questo caso il primo processo esegue il comando **cat** che concatena i tre file, il secondo esegue **grep** che li prende in ingresso e che restituisce le righe contenenti la parola *tree*.

In figura 3.2 si vedono i tre stati in cui si può trovare un processo:

1. Running.
2. Ready.
3. Blocked.

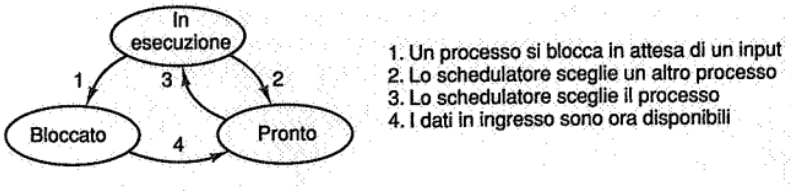


Figure 3.2: i possibili stati di un processo

Fra questi stati sono evidenziate le quattro possibili transizioni. La seconda e la terza sono causate dallo **scheduler** (che è una parte del sistema operativo) senza che i processi se ne accorgano. La quarta si verifica all'occorrere di un evento esterno come l'arrivo di un dato che un processo stava attendendo.

Fun fact: con il comando *ps* in UNIX è possibile vedere i processi terminati che per qualche motivo sono ancora gestiti dallo scheduler, questi hanno indicato nel campo *STAT* la dicitura **Z+**, essa sta per *zombie*, ovvero processo defunto.

Un altro modo di vedere la questione è mostrato in figura 3.3, che mostra più in dettaglio i diversi tipi di situazioni nelle quali si può trovare un processo.

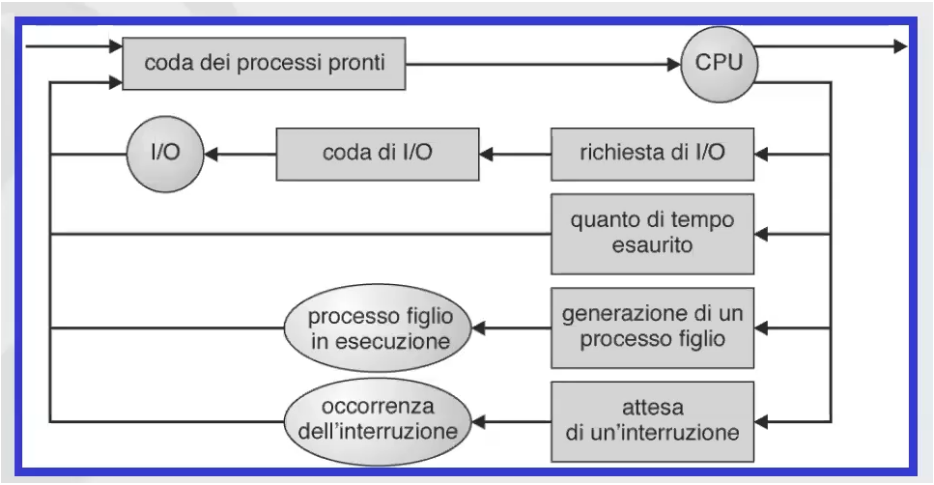


Figure 3.3: diagramma delle code

3.6 Implementazione

Per implementare il modello a processi il sistema operativo mantiene una tabella, detta **tabella dei processi**, in cui ciascun elemento referenzia un processo (gli elementi sono a volte chiamati **PCB** - Process Control Block). I PCB contengono le informazioni sullo stato del processo, il suo program counter, lo stack pointer, l'allocazione di memoria, lo stato dei suoi file aperti, le informazioni necessarie per la schedulazione e qualunque altra informazione utile per passare tra lo stato *running* e gli altri stati, in maniera da far ripartire il processo al punto di interruzione, come se non si fosse mai fermato.

Un processo UNIX spesso viene schematizzato dalle sue 4 componenti principali:

1. **u-area**: dati relativi al processo di pertinenza del sistema operativo (tabella dei file, working directory...).
2. **dati**: dati globali del processo.
3. **stack**
4. **codice**: il codice eseguibile.

La figura 3.4 mostra alcuni dei campi più importanti di un blocco di controllo di un sistema tipico.
La figura 3.5 invece riassume la gestione dell'interruzione e la schedulazione.
Infine la figura 3.6 mostra un tipico **context switch**.

3.7 Miscellanea

3.7.1 $PATH

In UNIX la shell non ha bisogno del percorso dei programmi che gli vengono chiesti di eseguire, essa cerca nei percorsi indicati dalla variabile d'ambiente *$PATH*.

3.7.2 La famiglia exec

Esistono diversi comandi exec: **execl**, **execle**, **execlp**, **execv**, **execvp**, **execvpe**. I caratteri aggiuntivi sono:

- **l**: passi una lista di argomenti invece di un vettore (l'ultimo elemento deve essere NULL).
- **e**: dopo la lista passi l'environment (variabili d'ambiente).
- **p**: passi, come primo parametro, la variabile di shell *$PATH*.
- **v**: passi con un vettore anziché una lista (ultimo elemento sempre NULL).

L'unica effettiva chiamata di sistema è **execve**. Le altre sono usate per facilitare il programmatore.

Gestione dei processi	Gestione della memoria	Gestione dei file
Registri	Puntatore al segmento di testo	Directory radice
Program counter	Puntatore al segmento dati	Directory di lavoro
Parola di stato dal programma	Puntatore al segmento dello stack	Descrittori dei file
Stack pointer		Identificatore dell'utente (UID)
Stato del processo		Identificatore del gruppo (GID)
Priorità		
Parametri di schedulazione		
Identificatore del processo		
Processo genitore		
Gruppo del processo		
Segnali		
Tempo al quale è stato fatto partire il processo		
Tempo di CPU usato		
Tempo di CPU dei figli		
Tempo del prossimo allarme		

Figure 3.4: alcuni dei campi di un elemento tipico della tabella dei processi

1. L'hardware mette sullo stack il program counter ecc.
2. L'hardware carica un nuovo program counter dal vettore delle interruzioni
3. Una procedura in linguaggio assembler salva i registri
4. Una procedura in linguaggio assembler prepara il nuovo stack
5. Viene eseguita una procedura di servizio per le interruzioni scritta in C (solitamente, legge i dati in ingresso e li mette in un buffer)
6. Lo schedulatore decide qual è il processo successivo da mandare in esecuzione.
7. La procedura in C restituisce il controllo al codice assembler.
8. La routine in codice assembler manda in esecuzione il processo corrente.

Figure 3.5: schema delle azioni del livello più basso del sistema operativo quando si verifica un interruzione

3.7.3 La funzione system

Un'alternativa alla famiglia *exec* prevede di utilizzare il comando **system**, esso accetta come parametro una stringa, che esegue in una shell come se l'utente l'avesse digitata in un terminale.

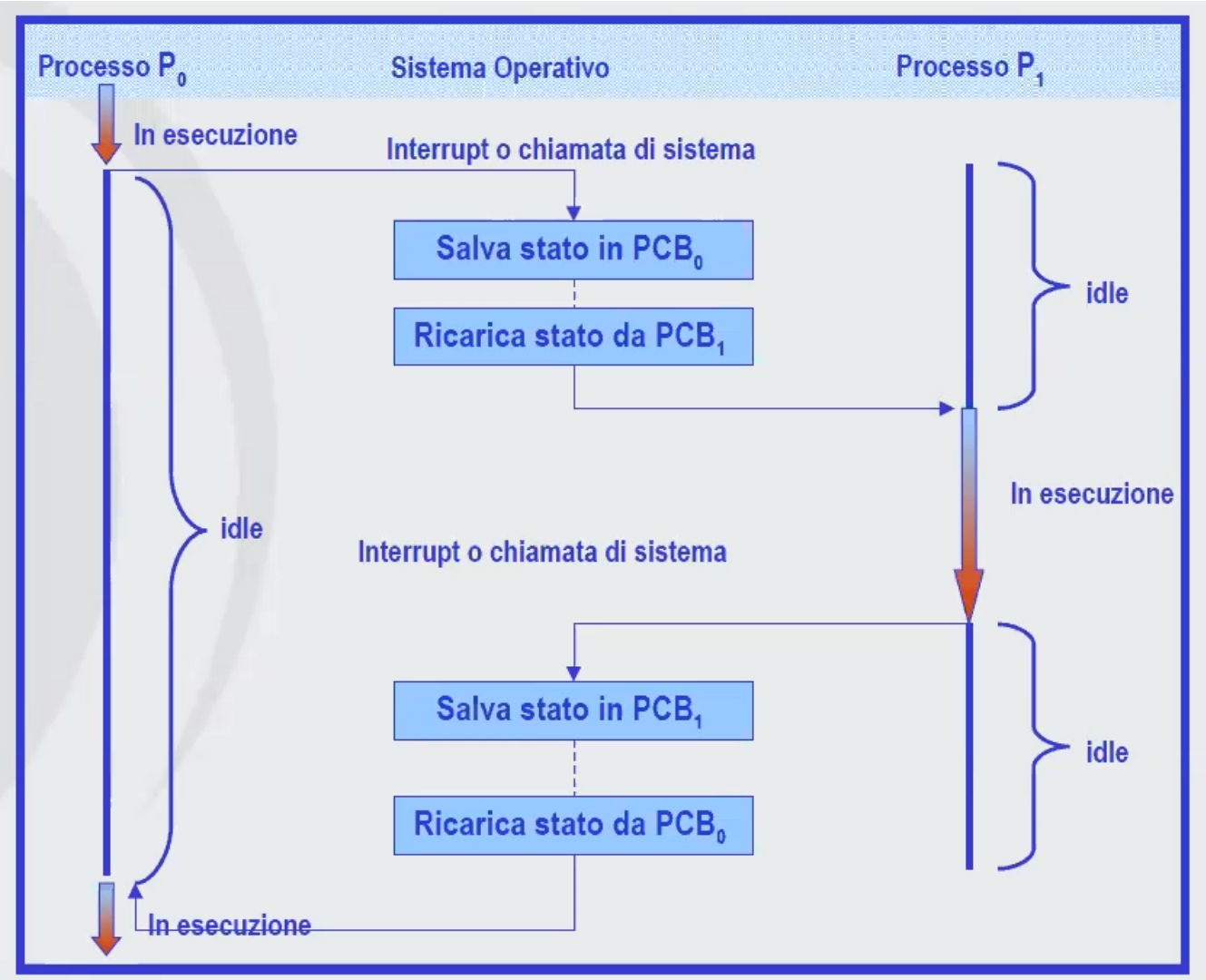


Figure 3.6: esempio di cambio di contesto

La schedulazione

Quando sono nello stato di pronto i processi competono per l'uso della CPU. La parte del sistema operativo che si occupa di scegliere quale processo eseguire si chiama **scheduler**.

Il **dispatcher** è il modulo che dà il controllo della CPU al processo selezionato dallo scheduler.

4.1 Introduzione

Oltre alla scelta lo schedulatore si occupa di far eseguire in modo efficiente la CPU, infatti il **context switch** è un operazione costosa: passaggio alla modalità kernel, salvataggio stato del processo (registri, memoria ...), scelta del processo successivo, caricamento stato del nuovo processo ed esecuzione di quest'ultimo. Spesso inoltre il *cambio di contesto* invalida l'intera cache, forzandone il ricaricamento.

Il tempo necessario è chiamato **latenza di dispatch** e dovrebbe essere reso il più piccolo possibile.

4.1.1 Comportamento

I processi in genere alternano raffiche di calcoli con richieste di operazioni I/O. Alcuni processi spendono la maggior parte del tempo in calcoli, essi sono chiamati **CPU bound**(*compute bound/orientati ai calcoli*). Altri invece sono spesso in attesa di operazioni di ingresso e uscita, loro sono chiamati **I/O bound**.

4.1.2 Quando schedulare

Esistono diverse situazioni in cui è utile schedulare, per esempio:

1. Creazione di un nuovo processo, si esegue genitore o figlio? Sono entrambi pronti.
2. Terminazione di un processo, si deve scegliere dalla lista di processi pronti (se non c'è nessuno si esegue il *processo inattivo* fornito dal sistema).
3. Blocco del processo corrente, per esempio per una richiesta I/O oppure a causa di un *semaforo*.
4. Interrupt I/O.

Più in generale si deve prendere una decisione ad ogni interruzione di clock, oppure dopo n interruzioni. In base a come gli **algoritmi di schedulazione** rispondono all'interrupt di clock vengono nominati:

- **nonpreemptive** (senza prerilascio), lascia in esecuzione il processo corrente fino a che non si blocca, non esegue controlli ad ogni clock.

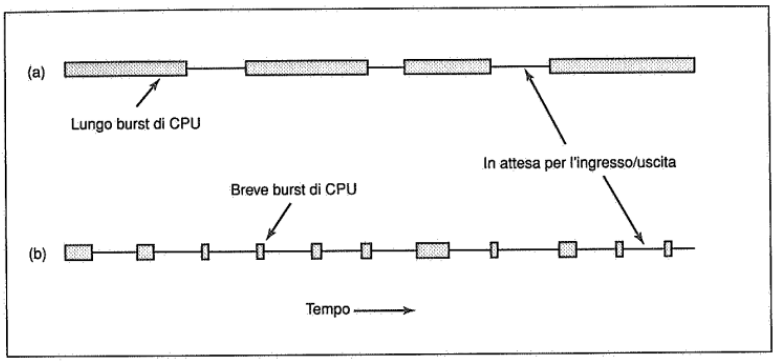


Figure 4.1: esempi di processi (a) CPU bound e (b) I/O bound

- **preemptive** (con prerilascio), lascia in esecuzione il processo per una quantità prefissata di tempo.

Nel secondo caso lo scheduler prende il controllo alla fine di ogni intervallo di tempo. Se non ci sono clock l'unica possibile soluzione è la prima.

4.1.3 Tipi di algoritmi

Lo scheduler deve ottimizzare cose diverse in base al sistema in questione. In generale riusciamo a dividere i possibili ambienti in tre:

1. **Batch**: non ci sono utenti impazienti.
2. **Interattivi**: qui ci sono utenti impazienti, il prerilascio è essenziale.
3. **Real time**: a volte il prerilascio non è necessario, spesso i processi in questo ambiente sono costruiti con l'intento di durare il meno possibile. A differenza degli ambienti interattivi qui si eseguono solo programmi che agevolano le applicazioni.

4.1.4 Obiettivi

La maggior parte degli obiettivi dipende dall'ambiente, ma ce ne sono alcuni che sono spesso opportuni.

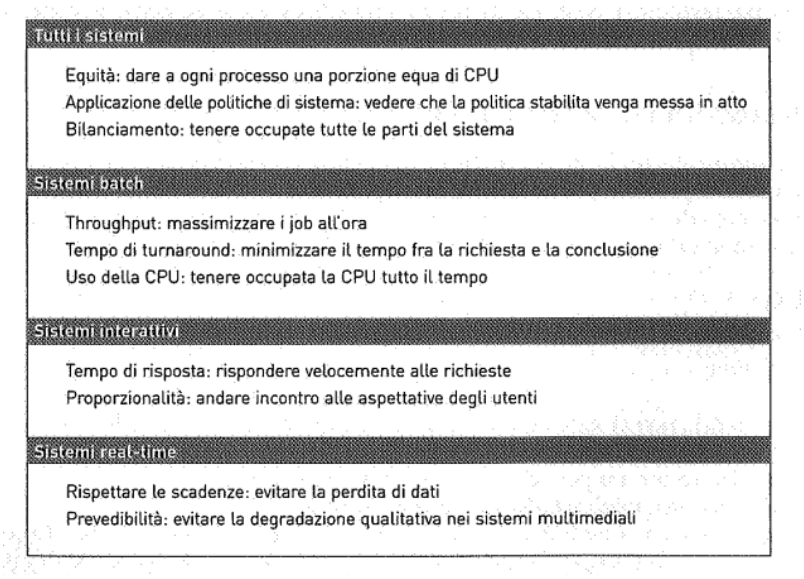


Figure 4.2: obiettivi dei diversi sistemi

Un generico obiettivo è quello di tenere la CPU il più occupata possibile, idem per i dispositivi di ingresso/uscita. I gestori di grossi centri di calcolo che eseguono molti job batch di solito considerano tre metriche per vedere quanto lavorano bene i loro sistemi:

1. **throughput**: il numero di job per ora che il sistema completa.
2. **turnaround**: tempo medio dal momento che viene richiesto un job batch fino a che non viene completato.
3. **uso della CPU**: parametro abbastanza inutile, è come valutare le auto basandosi sui giri all'ora del motore.

Per i sistemi interattivi, in particolar modo *timesharing* e *server*, è importante minimizzare il **tempo di risposta**, ovvero il tempo che intercorre tra invio di un comando e arrivo del risultato.

4.2 Schedulazione nei sistemi batch

Alcuni algoritmi sono adeguati sia a sistemi batch che interattivi, di seguito ci concentriamo su quelli adatti ai sistemi batch.

4.2.1 First-come First-served

Senza prerilascio.

Il primo che arriva viene servito, di seguito gli altri. C'è una coda di processi nello stato di pronto. Facile da capire e da programmare, equo.

4.2.2 Shortest Job First

Senza prerilascio.

Occorre conoscere in anticipo i tempi di esecuzione. Quando nella coda si trovano una serie di job di uguale importanza lo scheduler, usando lo *SJF* esegue il più breve.

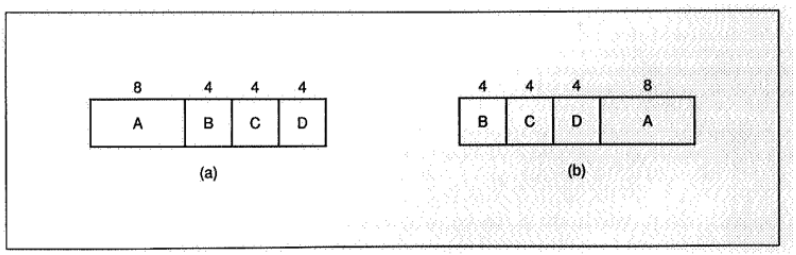
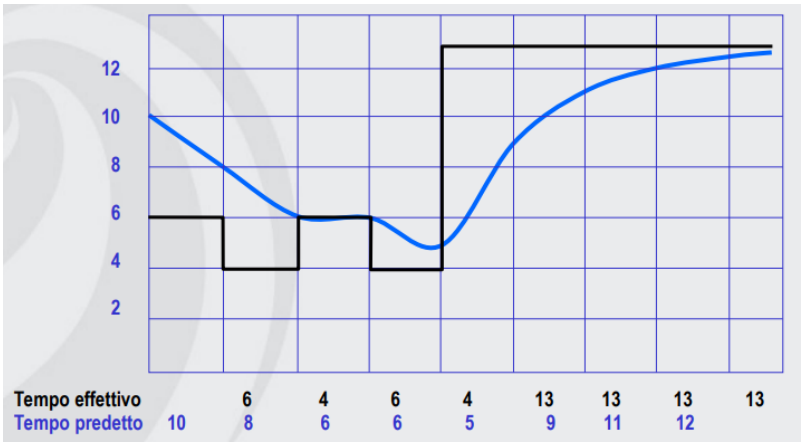


Figure 4.3: coda di processi dello *SJF*

Una grossa pecca di questo algoritmo è la necessità di avere tutti i job disponibili contemporaneamente, se un processo breve arriva un istante dopo l'esecuzione di uno lungo si perdono i vantaggi di questo algoritmo.

Un'alternativa al conoscere in anticipo i tempi di esecuzione prevede di stimarli. Per stimare il valore successivo di una serie si utilizza la tecnica dell' **aging**(invecchiamento), essa si basa sul calcolo della media pesata del valore corrente misurato e la stima precedente:

$$T_{st} = \alpha T_{mis} + (1 - \alpha) T_{st} \tag{4.1}$$



Shortest Remaining Time

Una variante dello *SJF*. Viene eseguito il processo con il tempo **rimasto** più breve. Anche qui il tempo di esecuzione deve essere noto in anticipo.

4.3 Schedulazione nei sistemi interattivi

Vediamo alcuni algoritmi che possono essere usati nei sistemi interattivi, ma che sono utilizzabili anche nei sistemi batch.

4.3.1 Round Robin

E' uno degli algoritmi più vecchi, imparziali e semplici. Ad ogni processo viene assegnato un **quanto**, durante il quale può essere in esecuzione. Se alla fine del quanto ha ancora bisogno di essere eseguito la CPU viene comunque rilasciata ad un altro processo, allo stesso modo di come si farebbe se finisse la sua esecuzione (o andasse nella lista dei processi *blocked*) prima della fine del quanto. Lo schedulatore ha bisogno solo di una lista di processi eseguibili.

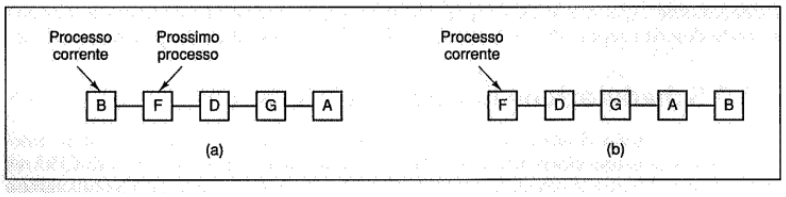


Figure 4.4

La questione interessante è la durata del quanto. Supponiamo che il *cambio di contesto* impieghi 1 millisecondo, se il quanto è di 4 millisecondi il 20% del tempo della CPU risulterà sprecato in overhead di gestione ($\frac{1}{1+4} = 20\%$). Aumentando il quanto la percentuale potrebbe drasticamente diminuire.

Nota che eliminare la prelazione migliorerebbe di parecchio le prestazioni, infatti in quel caso gli scambi di processi sarebbero eseguiti solo quando necessari, ovvero quando un processo si blocca.

Assegnare un quanto troppo breve provoca troppi cambi di contesto, peggiorando l'efficienza della CPU. Assegnare invece quanti troppo lunghi potrebbe provocare tempi di risposta esagerati per richieste interattive brevi.

4.3.2 Con priorità

Ad ogni processo viene assegnata una priorità, viene eseguito il processo che ha la più alta. Per evitare che alcuni processi ad alta priorità rimangano in esecuzione per un tempo indefinito la priorità viene diminuita ad ogni interrupt di clock,. In alternativa è possibile assegnare un quanto di tempo prefissato come limite massimo, oltre il quale il processo viene bloccato e ne viene mandato un altro in esecuzione, in base alle priorità.

Le priorità possono essere assegnate staticamente o dinamicamente.

```
Nota:
In UNIX "nice" permette ad un utente di diminuire volontariamente la priorità del suo processo.
```

Un semplice algoritmo, che rende un buon servizio ai processi *I/O bound*, prevede di assegnare priorità uguale ad $1/f$, dove f è la frazione di quanto che un processo ha usato.

Spesso si utilizzano in combinazione i metodi: si assegna una priorità alle classi di processi, e si usa la schedulazione round robin o altre all'interno di ciascuna classe.

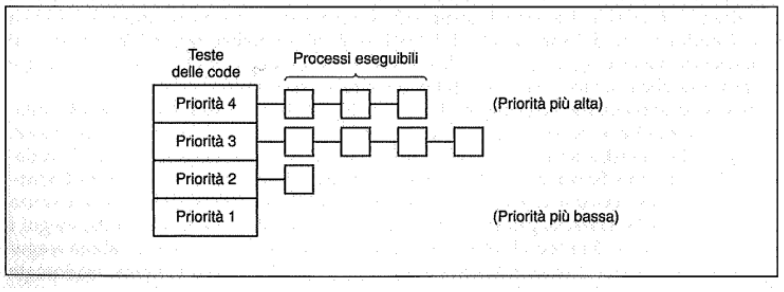


Figure 4.5

Un esempio è mostrato in figura 4.5. Va notato che se le priorità non vengono aggiornate di sovente, le classi di priorità più bassa possono aspettare troppo tempo.

4.3.3 A code multiple

Questo algoritmo prevede ancora una volta una suddivisione in classi di priorità: i processi appartenenti alla prima vengono eseguiti per un quanto, quelli della seconda per due, quelli della terza per quattro e così via ...

Ogni volta che un processo viene eseguito per l'intero suo quanto viene spostato ad una classe superiore. Man mano che un processo sale esso verrà eseguito sempre con minor frequenza, risparmiando tempo di CPU per i processi interattivi brevi.

Ogni coda può usare un algoritmo di scheduling personale. Va fatto uno scheduling anche tra le code.

Per prevenire che un processo inizialmente poco interattivo (e in seguito molto di più) venga punito per sempre sono adottate delle scappatoie, come il reset della priorità dei processi in cui viene premuto il tasto *invio*.

4.3.4 Schedulazione garantita

In questo caso l'approccio è completamente diverso dagli altri, si fa all'utente una promessa, e la si mantiene. Per esempio, se ci sono n utenti connessi si promette all'utente che riceverà $1/n$ della potenza della CPU. Stessa cosa vale per i processi.

Per poter adottare questo metodo lo scheduler ha bisogno di tenere traccia di quanta CPU utilizza ciascun processo e rapportare l'informazione con la promessa fatta. Viene definito il *rapporto* di un processo, semplicemente come il rapporto tra tempo di utilizzo della CPU effettivo fratto il tempo di CPU previsto, viene quindi eseguito il processo con il rapporto più basso.

4.3.5 Schedulazione a lotteria

L'idea di base è fornire ai processi dei biglietti della lotteria per varie risorse del sistema, come il tempo di CPU. Ogni volta che deve essere presa una decisione di schedulazione viene scelto a caso un biglietto della lotteria, premiando il processo che lo possiede.

La parte furba della soluzione consiste nel dare più biglietti ai processi più importanti. A lungo andare un processo che detiene una percentuale p di biglietti userà una percentuale p della risorsa.

Un altro punto importante è che processi cooperativi si possono scambiare i biglietti a piacimento.

4.4 Schedulazione nei sistemi real time

I sistemi real time si dividono in:

- **hard real time systems**
- **soft real time systems**

Nei primi è imperativo rispettare le scadenze, nei secondi anche ma è consentita una certa flessibilità all'errore.

In ogni caso il comportamento viene ottenuto dividendo il programma in processi, il cui comportamento è prevedibile e noto in anticipo. Questi processi hanno spesso vita breve e vengono eseguiti nella loro interezza in brevissimo tempo..

Gli eventi di un sistema real time possono essere **periodici** o **aperiodici**.

Nel caso in cui il sistema debba interagire con molti eventi periodici è necessario controllare che la gestione sia fattibile: ad esempio, se ci sono m eventi periodici, l'evento i-esimo arriva con periodo P_i e richiede C_i secondi di tempo di CPU per essere gestito, il carico può essere gestito solo se $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$.

Un sistema real time che rispetta questo vincolo è detto **schedulabile**.

In ultimo va sottolineato che i sistemi real time possono essere statici o dinamici, in base a quando prendono le decisioni: nel primo caso le decisioni vengono prese prima che il sistema inizi l'esecuzione. Nel secondo ad esecuzione iniziata. Ovviamente nel primo caso sono necessarie conoscenze a priori.

4.5 I livelli di scheduling

Fino ad ora abbiamo dato per scontato che la memoria principale fosse sufficiente per mantenere tutti i processi da schedulare. Ma che succede se così non fosse?

In questo caso lo scheduling dei processi comporta tempi di switch molto diversi fra processi in memoria principale e di massa.

Un modo pratico di gestire la situazione prevede di utilizzare uno scheduler **a livelli**. Esso è diviso in:

- scheduler a breve termine: identico a quello visto fino ad ora.
- scheduler a medio termine: si occupa degli spostamenti dei processi tra memoria e disco, rimuovendo quelli che sono stati in memoria principale abbastanza e caricando quelli necessari dal disco.

- scheduler a lungo termine: sceglie quali job mandare in esecuzione.

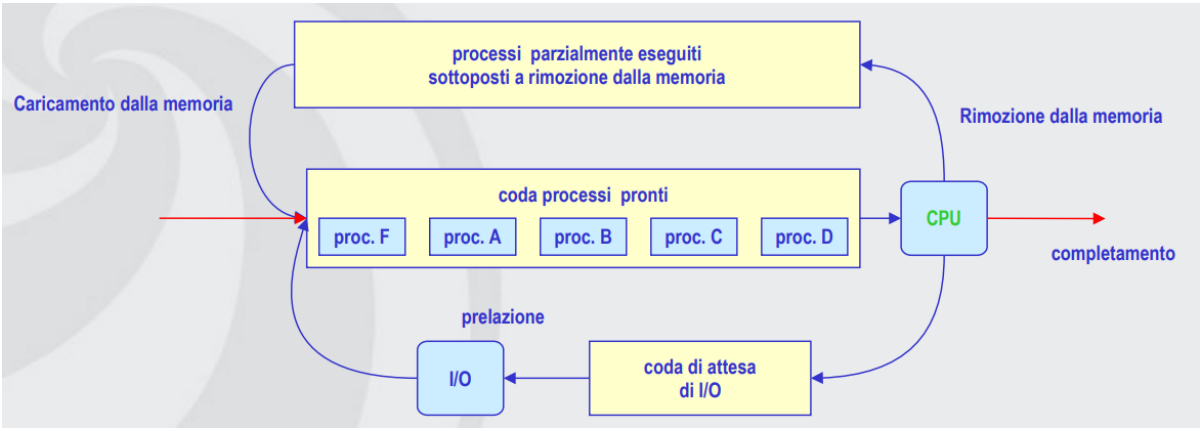


Figure 4.6: scheduling a medio termine

I criteri utilizzati dallo scheduler di medio termine sono:

- il tempo passato dall'ultimo spostamento in memoria.
- tempo di CPU assegnato al processo in questione.
- grandezza del processo.
- priorità del processo.

L'algoritmo utilizzato può essere uno di quelli già presentati.

4.6 Linux e gli altri..

Nel kernel Linux 2.6 esistono 140 livelli di priorità, quelli da 0 a 99 sono assegnati ai processi cosiddetti *real time* (non sono realmente real time), e quelli da 100 a 139 per i processi normali.

La **static priority** si calcola come SP $= 120 +$ *nice*. Il quanto di tempo invece vale $(140 - SP) * 20$ *ms* se SP è minore di 120, altrimenti $(140 - SP) * 5$ *ms*.

Per salvare le informazioni relative ai processi si utilizzano strutture apposite (**runqueue** e **prioarray**).

Lo scheduling in Linux kernel 2.4 è diverso da quello presentato, esso sfrutta un concetto simile alla schedulazione a lotteria, usando dei crediti.

In Windows 2000 se ne usa un altro ancora, inoltre in esso i livelli sono 32 e la priorità più alta la hanno i processi dei livelli più alti.

4.7 Esercizi

...

Thread

Nei sistemi operativi tradizionali ogni processo ha uno spazio di indirizzamento e un singolo **thread** di controllo. Ci sono tuttavia situazioni in cui è desiderabile avere molti thread di controllo nello stesso spazio di indirizzamento, in esecuzione in parallelo come se fossero processi separati.

5.1 Il modello a thread

Il modello a processi visto finora è basato su due concetti indipendenti tra loro: raggruppamento delle risorse ed esecuzione. Qualche volta è utile separarli, ed è qui che entrano in gioco i thread.

Un concetto che un processo racchiude è quello di thread di esecuzione, normalmente abbreviato con thread. Il *thread* ha un program counter che tiene traccia della prossima istruzione da eseguire, ha dei registri, uno stack...

I processi sono usati per raggruppare le risorse, i thread invece sono le entità schedulate per l'esecuzione nella CPU. Quello che i thread aggiungono al modello a processi è il permettere molte esecuzioni nell'ambiente di un processo, indipendenti l'una dall'altra.

Poiché i thread hanno solo alcune proprietà dei processi vengono a volte chiamati *processi leggeri*. Il termine **multithreading** è usato per descrivere quella situazione in cui ad un processo sono assegnati più thread.

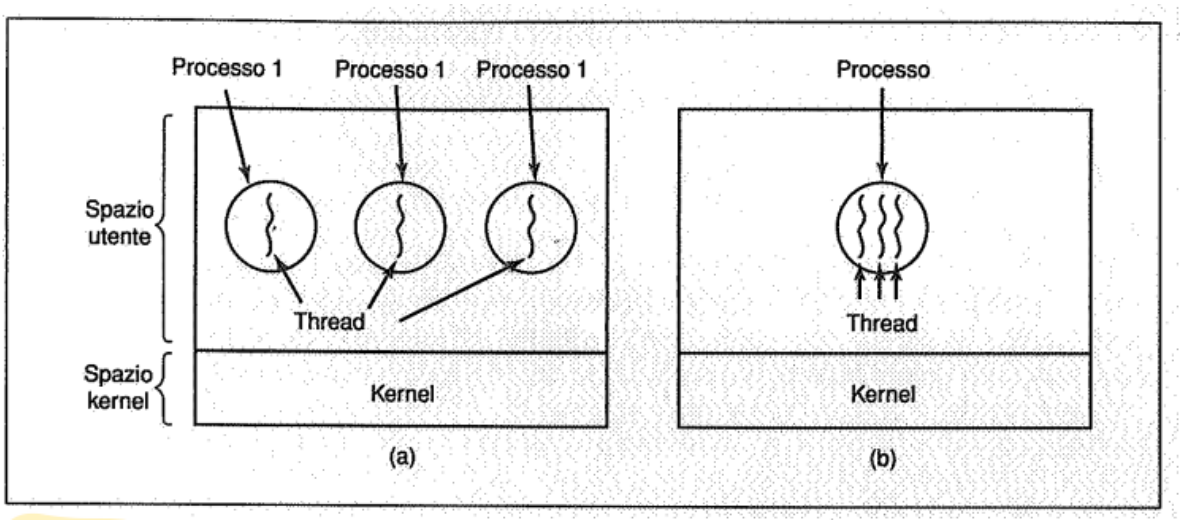


Figure 5.1

In entrambi i casi presentati dalla figura 5.1 abbiamo tre thread, nel primo caso ciascuno opera in uno spazio di indirizzamento diverso dagli altri, nel secondo caso lo spazio è condiviso. COnviene utilizzare la seconda soluzione nel caso in cui i thread sono veramente parte dello stesso job e cooperano l'un l'altro attivamente.

Quando un processo con thread multipli viene eseguito la CPU passa rapidamente tra uno a l'altro, dando l'impressione di un esecuzione parallela.

Rispetto che per i processi, per i thread non esiste protezione, non è necessaria e sarebbe comunque impossibile da realizzare.

Oltre che lo stesso spazio di indirizzamento i thread condividono: lo stesso insieme di file aperti, processi figli, allarmi, segnali...

Voci relative a ciascun processo	Voci relative ad ogni thread
Spazio di indirizzamento	Program counter
Variabili globali	Registri
File aperti	Stack
Processi figli	Stato
Allarmi incombenti	
Segnali e gestori di segnali	
Informazioni di contabilità	

Figure 5.2

Come un processo tradizionale (ovvero dotato di un singolo thread) anche i thread possono trovarsi in uno qualsiasi dei diversi stati: pronto, esecuzione, bloccato, terminato. Vale la stessa cosa per le transizioni degli stati, sono analoghe a quelle dei processi.

È importante capire che ogni thread ha il proprio stack, esso contiene un elemento per ogni procedura chiamata non ancora conclusa, con le variabili locali della procedura e gli indirizzi di ritorno. Ogni thread chiamerà generalmente procedure diverse ed avrà quindi una diversa storia di esecuzione.

Quando è presente *multithreading* normalmente i processi partono con un solo thread, che ha la capacità di creare nuovi thread con una procedura di libreria, ad esempio **thread_create**, con un parametro indicante una procedura che il nuovo thread deve eseguire.

Qualche volta i thread sono gerarchici, altre volte essi sono equivalenti tra loro. In entrambi i casi ad un thread che ne crea un altro viene restituito un identificatore di quest'ultimo.

Quando un thread ha finito chiama un'altra procedura di libreria, la **thread_exit**. In alcuni sistemi è possibile che un thread aspetti che uno specifico thread termini con la chiamata di procedura **thread_wait**.

Un'altra chiamata comune è la **thread_yield** che permette di cedere la CPU ad un altro thread, perché venga eseguito. È una chiamata importante in quanto non esiste un interruzione di clock per realizzare il desiderato timesharing.

Oltre a vantaggi i thread introducono anche delle complicazioni.. che fare quando un *processo* genitore ha thread multipli e chiama la *fork()*? Se il figlio ottiene tanti thread quanti il genitore che succede se un thread del genitore viene bloccato?

O ancora, che succede se un thread che condivide file aperti con altri chiude un file che un altro stava ancora leggendo?

Occorre prestare molta attenzione a come programmare i diversi compiti di ciascun thread.

5.2 Uso dei thread

I/O I thread possono essere creati e distrutti più facilmente dei processi, poiché non hanno alcuna risorsa associata. Essi non portano alcun guadagno quando sono tutti *CPU bound*, quando invece sono presenti molte operazioni *I/O* danno il loro meglio.

I thread sono utili anche per sistemi con CPU multiple, in cui è possibile un reale parallelismo.

Come esempio si può considerare un elaboratore di testo: utilizzando tre thread il modello di programmazione è molto più semplice, il primo può occuparsi dell'interazione con l'utente, il secondo della riformattazione del documento e il terzo che salva il contenuto della RAM su disco periodicamente. Si noti che avere tre processi che operano separatamente in questo caso non funzionerebbe.

In generale, in informatica, quando in un modello ogni computazione ha uno stato salvato ed esistono un insieme di eventi per permettere il cambio di stato, si parla di **macchina a stati finiti**.

Ciò che offrono i thread quindi è parallelismo anche in quei casi in cui l'idea del processo è sequenziale con chiamate di sistema bloccanti.

5.3 Implementazione dei thread

Ci sono due modi principali di implementare i thread: nello spazio utente e nel kernel.

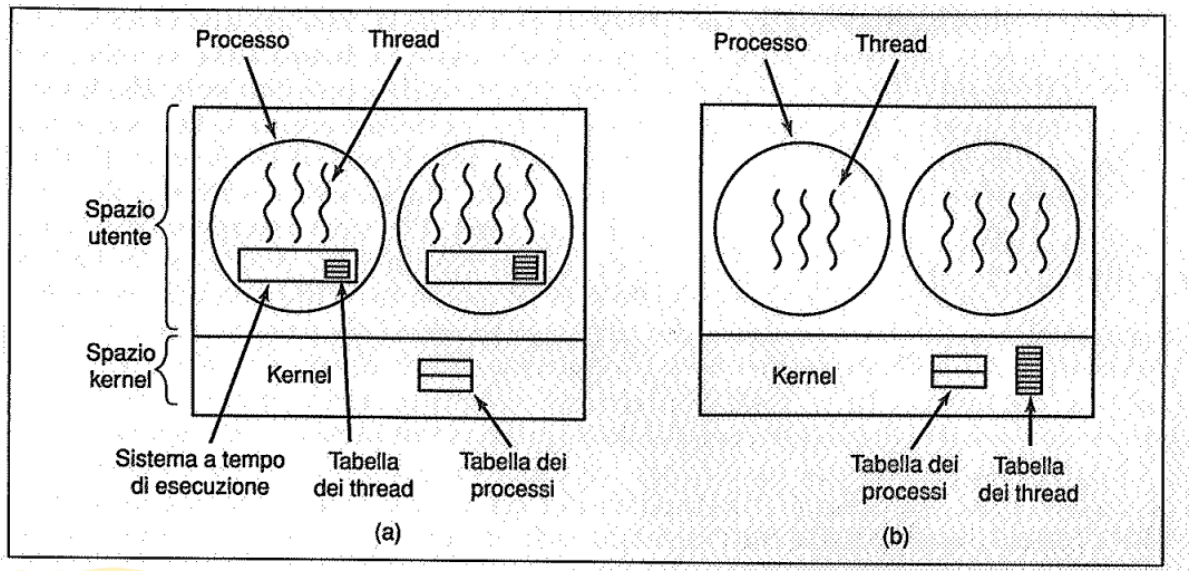


Figure 5.3: (a) thread package a livello utente. (b) thread package gestito dal kernel.

5.3.1 Nello spazio utente

In questo metodo i thread sono completamente nello spazio utente, il kernel non sa nulla di loro e gestirà i processi ordinari ad un solo thread.

Il primo vantaggio di questa soluzione è permettere l'implementazione dei thread anche in quei sistemi operativi che non li supportano nativamente.

Come si vede in figura 5.3(a) i thread sono eseguiti sopra ad un sistema a tempo di esecuzione, ovvero una collezione di procedure che li gestiscono.

In questa soluzione ogni processo ha bisogno della propria **tabella dei thread**, analoga a quella dei processi. Anch'essa è gestita dal sistema a tempo di esecuzione.

Quando un thread fa qualcosa che potrebbe bloccarlo localmente chiama una procedura a tempo di esecuzione, la quale controlla se il thread va effettivamente bloccato. In caso affermativo essa salva le informazioni utili nella tabella dei thread e ricarica i registri macchina con i valori salvati del nuovo thread. Non appena SP e PC vengono aggiornati parte l'esecuzione di quello nuovo senza la necessità di altre operazioni.

Scambiare thread in questo modo è almeno un ordine di grandezza più rapido che usare le trap del kernel, questo è un grande vantaggi dell'implementazione nello spazio utente. Esiste infatti una differenza chiave con i processi, la procedura che salva lo stato dei thread e lo schedulatore sono procedure locali, invocarle è quindi molto più efficiente che chiamare in causa il kernel.

Altri vantaggi di questo metodo sono: ogni processo può avere il proprio algoritmo di schedulazione, migliore scalabilità (rispetto a quelli implementati nel kernel non hanno bisogno di spazio per tabelle e stack direttamente nel kernel).

Uno degli svantaggi invece è l'implementazione delle chiamate di sistema bloccanti: se un thread legge da tastiera prima che un tasto sia premuto è impensabile lasciare che il thread esegua effettivamente la chiamata di sistema, bloccherebbe tutti i thread.

Nota: in alcune versioni UNIX esiste una scappatoia al problema: la chiamata di sistema **select** che permette di dire se una chiamata si bloccherà, prima di eseguirla. Nel caso si usi questa soluzione il codice intorno alla vera chiamata di sistema (che quindi effettuerà la *select*) è chiamato **wrapper**.

Un altro problema analogo è quello del *fault* di pagina, esso si verifica quando un thread fa un salto ad un area di memoria non caricata in RAM, in quel caso il sistema operativo blocca tutto per andare a caricare in RAM la porzione richiesta, compresi i thread che non c'entrano nulla, in quanto non ne conosce l'esistenza.

Un ultimo problema si ha nella gestione stessa: quando un thread inizia l'esecuzione nessun altro thread in quel processo verrà eseguito, a meno che il primo non rilasci spontaneamente la CPU. Se un thread non entra nel sistema a tempo di esecuzione lo schedulatore non più far nulla.

La gestione dei thread a livello utente è fatta tramite librerie:

- POSIX Pthreads
- Java threads
- C-threads (Mach)
- API di win32

Un semplice programma che usa la libreria Pthreads:

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

void *test_thread(void *pt)
{
    printf("thread: %s\n", (char *)pt);
    pthread_exit(pt);
}

void main()
{
    pthread_t t;
    pthread_attr_t attr;
    void *res;
    char msg[6] = "hello";

    pthread_attr_init(&attr);

    /*
    * Crea un nuovo thread, simile ad una fork() seguita da una exec()
    *
    * 1 la struttura thread.
    * 2 la struttura attributi.
    * 3 puntatore a funzione che eseguira' il thread.
    * 4 argomento della funzione.
    * @return status
    */
    if(pthread_create(&t, &attr, test_thread, &msg[0]))
        exit(1);

    /*
    * Aspetta che il thread finisca, analogo di waitpid()
    *
    * 1 il thread.
    * 2 variabile d'appoggio per la comunicazione con il thread figlio.
    */
    pthread_join(t, &res);
    if(res)
        printf("%s\n", (char *)res);
    return;
}
```

5.3.2 Nel kernel

Se il kernel conosce e gestisce i thread non serve un sistema a tempo di esecuzione. Questa situazione è mostrata in figura 5.3(b). Non c'è una tabella dei thread per ogni processo ma ne esiste una globale.

Quando un thread vuole ne vuole creare un altro esso esegue una chiamata di sistema che aggiornerà la tabella dei thread. La tabella conterrà le stesse informazioni di prima ma sarà salvata nel kernel.

Tutte le chiamate che possono bloccare un thread sono chiamate di sistema, hanno quindi un costo decisamente più elevato, tuttavia il kernel può decidere di far continuare l'esecuzione di un altro thread appartenente al processo (oppure di un altro processo) nel frattempo che il thread risulta bloccato.

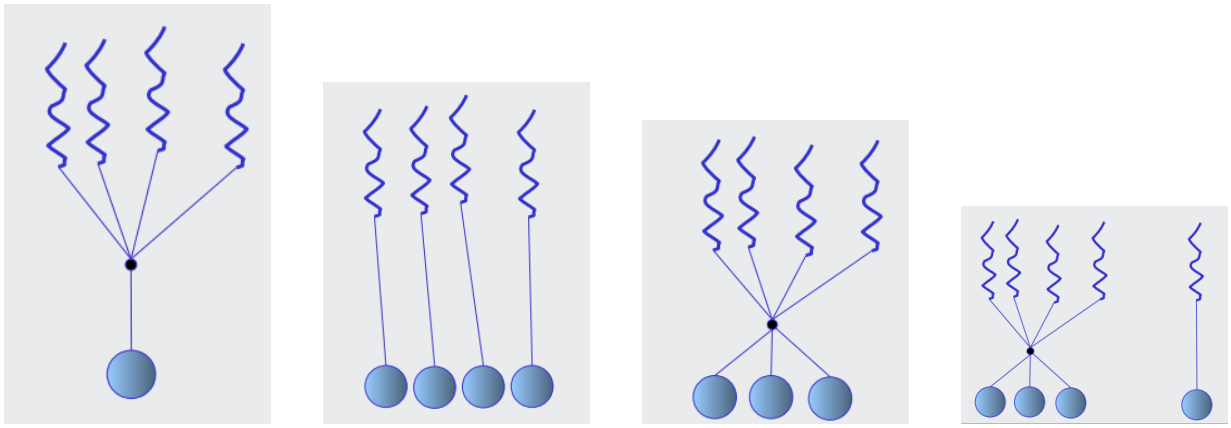
Per sopperire un po' all'eccessivo costo di creazione e distruzione dei thread alcuni sistemi implementano strategie ad hoc che consistono nel riciclaggio dei thread, questo permette di ridurre il grande *overhead* causato dal problema.

Esiste un problema aggiuntivo e riguarda la schedulazione: se il processo A ha 1 thread ed il processo B ne ha 100, se implementati a livello utente i thread di B ottengono un centesimo del tempo di CPU del thread di A, se invece vengono implementati nel kernel a ciascun thread viene assegnato lo stesso tempo di CPU, ma a questo punto il processo A ottiene un centesimo rispetto al processo B...

5.4 Programmazione multithread

Esistono diversi approcci nella programmazione multithread:

- **molti a uno**: in questo caso molti thread a livello utente sono mappati in un singolo thread del nucleo. [*solaris green thread* e *GNU portable thread*]
- **uno a uno**: ad ogni thread a livello utente ne corrisponde uno nel nucleo. [*windows NT/XP/2000*, *linux*, *solaris 9+*]
- **molti a molti**: molti thread a livello utente mappati in molti thread nel kernel. Permette al sistema operativo di creare un numero sufficiente di thread a livello del nucleo. [*solaris 9-*, *windows NT/2000(lib. ThreadFIber)*]
- **a due livelli**: simile al molti-a-molti ma permette ad un thread utente di essere associato ad un thread del nucleo. [*irix*, *hp-ux*, *tru64 unix*, *solaris 8-*]



(a) molti a uno (b) uno a uno (c) molti a molti (d) a due livelli

Figure 5.4

5.5 Unix

I segnali in UNIX sono usati per comunicare ad un processo il verificarsi di un particolare evento.

Lo schema si ripete:

1. il segnale viene generato da un evento.
2. il segnale viene inviato ad un processo.
3. il segnale viene gestito:
   - viene inviato al thread a cui si riferisce.
   - ogni thread riceve lo riceve.
   - solo alcuni thread lo ricevono.
   - esiste un unico thread che li gestisce tutti.

Nota che nella terminologia Linux si parla di **task** invece che thread. La loro creazione avviene tramite la chiamata di sistema *clone()*, che permette al task figlio di condividere lo spazio di indirizzi del task genitore. La *fork()* diventa allora solo un caso particolare della *clone()*.

5.6 I thread di java

...

Comunicazione tra processi

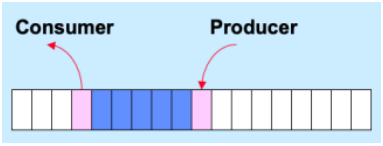
Occorre che i processi comunichino. Si parla di **IPC** - InterProcess Communication.

Ci sono tre questioni da considerare:

- passaggio di informazioni.
- concorrenza nelle sezioni critiche.
- sequenzializzazione in presenza di dipendenze.

Le ultime due questioni sono applicabili anche ai thread.

Un famoso problema è quello del **produttore-consumatore** (*bounded buffer problem*): due processi condividono un buffer comune di dimensione fissa, il produttore inserisce le informazioni, il consumatore le preleva. Il buffer ha due indici, uno per inserimento e uno per prelievo. Il problema che si viene a creare è la race condition, che può portare ad un deadlock, vedremo in seguito che significa.



6.1 Corse critiche

In alcuni sistemi i processi possono condividere una parte di memoria comune, che ciascuno può leggere e scrivere.

Situazioni nelle quali due processi stanno leggendo o scrivendo qualche dato condiviso, ed il risultato finale dipende dall'*ordine* con cui vengono eseguiti i processi, prendono il nome di **race conditions**, o *corse critiche*.

6.2 Sezioni critiche

Come evitare corse critiche? Abbiamo bisogno di **mutua esclusione**, un qualche modo per assicurarci che se un processo sta utilizzando una variabile od un file condivisi, gli altri processi saranno impossibilitati a fare lo stesso.

La parte di un programma che utilizza risorse condivise prende il nome di **sezione critica**, o *regione critica*.

Per avere una buona soluzione dobbiamo soddisfare quattro condizioni:

1. Due processi non devono mai trovarsi contemporaneamente nelle loro sezioni critiche.
2. Non si deve fare alcuna ipotesi sulle velocità e sul numero di CPU.
3. Nessun processo in esecuzione fuori dalla sua sezione critica può bloccare altri processi.
4. Nessun processo deve aspettare indefinitamente per poter entrare nella sua sezione critica.

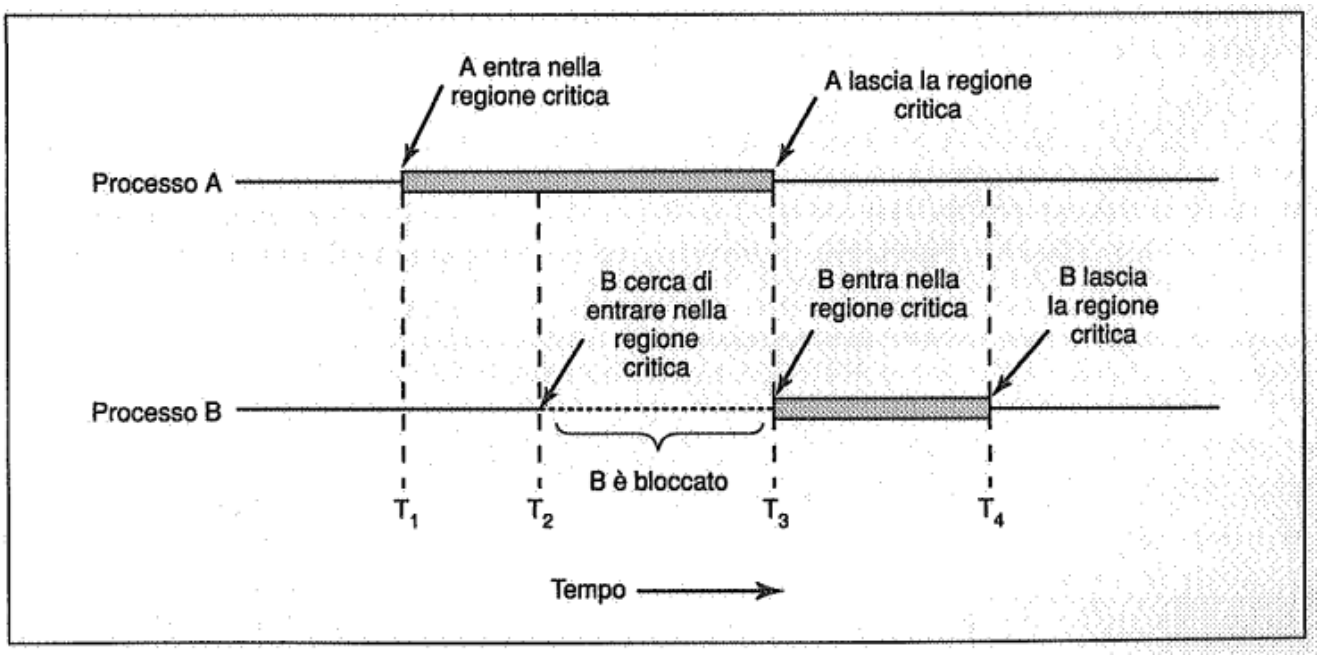


Figure 6.1: Il comportamento desiderato, mutua esclusione

6.3 Mutua esclusione con attesa attiva

6.3.1 Disabilitazione degli interrupt

La soluzione più semplice consiste nel permettere a ciascun processo di disabilitare le interruzioni non appena entra nella sua regione critica, e di riabilitarle non appena ne esce. Con le interruzioni disabilitate la CPU non può essere assegnata ad un altro processo.

Questa tecnica è spesso utile nel kernel ma è poco appropriata per i processi utente.

6.3.2 Variabili di lock

Supponiamo di avere una singola variabile condivisa (**di lock**), con valore iniziale 0. Quando un processo vuole entrare in regione critica controlla prima la variabile, se è 0 la imposta ad 1 ed entra, altrimenti aspetta.

Questa soluzione non è propriamente una soluzione in quanto può verificarsi lo stesso che due processi siano nelle loro sezioni critiche, basti pensare che la lettura della variabile di lock da parte di un processo, può essere immediatamente seguita dalla lettura della stessa da parte dell'altro processo, causando l'inevitabile.

6.3.3 Alternanza stretta

```
while (TRUE) {
   while (turno != 0) / * ciclo * / ;
   regione_critica( );
   turno = 1;
   regione_non_critica( );
}
```

(a)

```
while (TRUE) {
   while (turno != 1) / * ciclo * /;
   regione_critica( );
   turno = 0;
   regione_non_critica( );
}
```

(b)

Figure 6.2: (a) processo 0 (b) processo 1.

Nell'approccio presentato in figura 6.2 la variabile intera *turno* tiene traccia del processo al quale tocca.

Il testare continuamente una variabile, in attesa che essa assuma un certo valore, è detta **attesa attiva**, o *busy waiting*. Di norma è una procedure che andrebbe evitata, visto l'incredibile spreco di CPU. Un lock che usa l'attesa attiva è detto **spin lock**.

Questa soluzione è cattiva se uno dei due processi è molto più lento dell'altro. Inoltre anche questa non è una vera e propria soluzione: infatti essa viola la condizione 3 enunciata prima, il processo 0 è bloccato da un processo che non è nella sua sezione critica.

Pur non essendo una soluzione rappresenta una modalità, poco seria, di evitare qualsiasi corsa critica.

6.3.4 La soluzione di Peterson

```
#define FALSE 0
#define TRUE 1
#define N 2                                    / * Numero di processi * /

int turno;                                     / * A chi tocca?* /
int interessato[N];                            / * Tutti i valori inizialmente 0 o FALSE * /

 void entra_nella_regione(int processo);       / * processo è 0 o 1* /
{
   int altri;                                  / * Numero dell'altro processo * /
   altri = 1 - processo;                       / * L'altro processo * /
   interessato[processo] = TRUE;               / * Mostra che sei interessato * /
   turno = processo;                           / * Imposta il flag * /
   while (turno == processo && interessato[altri] == TRUE) / * Istruzione nulla * /;
}

 void lascia_la_regione(int processo)          / * Processo: chi lascia la regione critica * /
{
   interessato[processo] = FALSE;              / * Uscita dalla regione critica * /
}
```

Figure 6.3: La soluzione di Peterson

L'algoritmo di Peterson è mostrato in figura 6.3.

Prima di entrare nella propria regione critica, ciascun processo chiama la funzione *entra_nella_regione(id)* con il proprio identificativo. Invece per indicare che ha finito chiama *lascia_la_regione(id)*, permettendo agli altri di entrare.

Consideriamo il caso peggiore: due processi chiamano *entra_nella_regione(id)* quasi contemporaneamente. Ciò che succede è che il processo che la chiama per ultimo vedrà impostato il proprio *id* nella variabile turno, quindi eseguirà la propria regione critica, mentre l'altro eseguirà il *while* a vuoto finché il primo non libererà la risorsa.

6.3.5 L'istruzione TSL

Questa soluzione è una proposta hardware. I calcolatori progettati pensando ai processi multipli implementano un istruzione particolare:

```
TSL RX,LOCK
```

Test and Set Lock, verifica ed imposta il blocco: mette il contenuto della parola di memoria *lock* nel registro *rx*, e in seguito memorizza un valore diverso da 0 all'indirizzo di memoria di *lock*. Queste due operazioni sono garantite indivisibili: nessun altro processore può accedere alla parola finché l'istruzione non è finita (essa è quindi un operazione **atomica**).

Per usare questa soluzione useremo una variabile condivisa, *lock*, per coordinare l'accesso: quando essa è a 0 qualsiasi processo la può mettere ad 1 con *TSL* e riportarla, in seguito all'esecuzione della sua regione critica, a 1 (usando una semplice istruzione *move*).

6.4 Sospensione e risveglio

Tutte le soluzioni viste finora hanno un problema: l'attesa attiva. Possono anche venirsi a creare effetti inaspettati: se un processo *a*, ad alta priorità, si trova nella sua regione critica, e uno *b*, con priorità più bassa, passa nello stato *ready*, il primo comincia l'attesa attiva, ma poiché *b* non viene mai scelto quando *a* è in esecuzione, *a* non ha mai modo di lasciare la regione critica e *b* cicla all'infinito. Questo è il **problema di inversione delle priorità**.

Vediamo alcune primitive di comunicazione tra processi che li bloccano, anziché sprecare tempo di CPU.

Una delle coppie più semplici è formata dalle primitive **sleep** e **wake up**: la prima provoca il blocco del processo chiamante, la seconda, tramite un parametro identificativo del processo in questione, lo risveglia. Esiste un'alternativa che specifica un parametro sia per *sleep* che per *wake up*, in modo da accoppiarle.

Per esprimere le chiamate di sistema corrispondenti le mostreremo come semplici chiamate a procedure di libreria in C. Non fanno parte della libreria standard, ma saranno presumibilmente disponibili in ogni sistema che le supporta.

Un classico problema che le coinvolge è una sveglia spedita ad un processo sveglio, in quel caso occorre aggiungere un **bit di attesa della sveglia** (wake up waiting bit), che non è altro che un salvadanaio per i segnali di sveglia. Questa semplice soluzione non basta per situazioni con n processi.

6.5 I semafori

Dijkstra suggerì di usare una variabile intera per contare il numero di sveglie salvate per l'uso futuro. Nella sua proposta venne introdotto un nuovo tipo di variabile, chiamata **semaforo**: se esso ha valore 0 non è stata salvata alcuna sveglia, se ha un valore positivo una o più sveglie sono pervenute.

Vennero proposte due operazioni:

1. **down**, analoga di sleep: se il numero di sveglie è maggiore di zero viene decrementato di uno il valore; se è zero il processo viene sospeso, in attesa di completare la *down*.

2. **up**, analoga di wake up: incrementa il valore del semaforo di uno; se uno o più processi erano sospesi su quel semaforo ne viene scelto uno da fare ripartire (anche a caso).

Controllare il valore, cambiarlo ed eventualmente sospendersi sono tutte operazioni eseguite in un'unica, indivisibile, **azione atomica**. Anche incremento semaforo e risveglio sono istruzioni indivisibili. Questa atomicità è essenziale.

6.5.1 Produttore-Consumatore con semafori

La soluzione è mostrata in figura 6.4. Normalmente *up* e *down* sono implementate come chiamate di sistema ed il sistema operativo, siccome sono operazioni da pochissime istruzioni, disabilita gli interrupt senza causare danni. Nel caso di CPU multiple il semaforo va protetto con TSL per garantire che solo una alla volta lo interroghi.

```
#define N 100                          / * Numero delle posizioni del buffer * /
typedef int semaforo;                  / * I semafori sono un tipo di interi * /
semaforo mutex = 1;                    / * Controlla l'accesso alla regione critica * /
semaforo vuoti = N;                    / * Conta le posizioni vuote nel buffer * /
semaforo pieni = 0;                    / * Conta le posizioni piene nel buffer * /

void produttore(void)
{
   int item;

   while (TRUE) {                      / * TRUE è la costante 1 * /
      item = produci_elemento( );      / * Genera qualcosa da inserire nel buffer * /
      down(&vuoti);                    / * Decrementa contatore posizioni vuote * /
      down(&mutex);                    / * Entra nella regione critica * /
      inserisci_elemento(item);        / * Metti il nuovo elemento nel buffer * /
      up(&mutex);                      / * Lascia la regione critica * /
      up(&pieni);                      / * Incrementa contatore posizioni piene * /
   }
}

void consumatore(void)
{
   int item;

   while (TRUE) {                      / * Ciclo infinito * /
      down(&pieni);                    / * Decrementa contatore posizioni piene * /
      down(&mutex);                    / * Entra nella regione critica * /
      item = estrai_elemento( );       / * Prendi un elemento dal buffer * /
      up(&mutex);                      / * Abbandona la regione critica * /
      up(&vuoti);                      / * Incrementa contatore posizioni vuote * /
      consuma_elemento(item);          / * Fai qualcosa con l'elemento * /
   }
}
```

Figure 6.4: Produttore-Consumatore con semaforo.

La soluzione prevede l'utilizzo di tre semafori:

1. *pieni*: conta il numero di elementi che sono occupati. Inizialmente posto a zero.

2. *vuoti*: conta quelli vuoti. Inizialmente posto uguale al numero di elementi nel buffer.

3. *mutex*: garantisce che produttore e consumatore non accedano contemporaneamente al buffer. Inizialmente posto a uno.

I semafori come *mutex*, che vengono inizializzati ad uno e sono usati per organizzare la mutua esclusione di due processi, sono detti **semafori binari**.

È garantita la mutua esclusione se ogni processo chiama una *down* prima di entrare nella sua regione critica, e una *up* subito dopo esserne uscito.

I semafori *pieni* e *vuoti* invece sono semafori di **sincronizzazione**, garantiscono quindi che l'*ordine* di alcune operazioni sia ben definito.

6.6 Mutex

Quando non è necessaria la caratteristica dei semafori di saper contare, ne viene utilizzata una versione semplificata, detta **mutex**. Sono usati per gestire la mutua esclusione, sono facili ed efficienti da implementare, vengono spesso usati per i thread implementati nello spazio utente (quando è prevista l'istruzione TSL).

Un *mutex* è una variabile che può essere in due stati: bloccato e non bloccato. Occorre quindi un solo bit per rappresentarlo, anche se spesso si utilizza un intero (zero significa bloccato, non bloccato per ogni altro valore).

Vengono usate due procedure:

1. *mutex lock*: simile a soluzioni già viste, ma con la sostanziale differenza che non implementa l'*attesa attiva*, non ci sarebbe alcun clock a fermare i thread in esecuzione da troppo. Quando un thread fallisce nell'acquisizione di un blocco chiama la **thread_yield**, per lasciare la CPU ad un altro thread (essa è una chiamata allo schedulatore non una di sistema).

2. *mutex_unlock*: il nome è autoesplicativo. Inoltre, se più thread sono bloccati sul mutex, ne viene scelto uno a caso per acquisire la risorsa.

Se i processi hanno spazio di indirizzamento separato, come abbiamo continuamente detto, come possono condividere la variabile turno dell'algoritmo di Peterson, o i semafori o un buffer comune?

Ci sono due risposte. Primo, alcune delle strutture dati condivise, come i semafori, possono essere memorizzate nel kernel e accedute solo tramite chiamate di sistema e questo approccio elimina il problema. Secondo, la maggior parte dei sistemi operativi moderni (compresi UNIX e Windows) fornisce un modo attraverso il quale i processi possono condividere porzioni del loro spazio di indirizzamento con altri processi, così i buffer e altre strutture dati possono essere condivise. Nel caso peggiore, quando non è possibile nient'altro, può essere usato un file condiviso.

6.7 Monitor

La scrittura di programmi contenenti semafori è tutt'altro che banale. Nel produttore-consumatore con semafori, se dovesse accadere che *mutex* venga decrementato prima di *vuoti* invece che dopo, se il buffer fosse completamente pieno, il produttore si bloccherebbe con *mutex* a 0, a questo punto alla prossima down del consumatore entrambi i processi risulterebbero bloccati per sempre... questa situazione viene spesso indicata con **deadlock** (stallo).

Per rendere più semplice la scrittura di questi programmi venne introdotta una primitiva di sincronizzazione più ad alto livello, chiamata **monitor**.

Un monitor è una collezione di procedure, variabili e strutture dati che vengono raggruppate insieme in un tipo speciale di modulo o package. I processi possono chiamare le procedure del monitor ma non possono accedere alle strutture interne.

Essi possiedono un importante proprietà che li rende utili per ottenere la mutua esclusione: ad ogni istante, un solo processo può essere attivo in un monitor. Essendo essi un costrutto del linguaggio di programmazione il compilatore sa che devono essere trattati con cautela. Tipicamente, ogni volta che una procedura del monitor va attivata, prima di farlo si controlla che nessun'altra sia in esecuzione, in caso affermativo si fa partire la procedura richiesta. Spetta quindi al compilatore implementare la mutua esclusione, sebbene sia un modo comune implementarla utilizzando un *mutex*.

Per chi scrive i programmi, a questo punto, basta sapere che, trasformando tutte le sezioni critiche in procedure del monitor, due processi qualunque non potranno mai eseguire contemporaneamente le proprie sezioni critiche.

Occorre definire come bloccare i processi quando non possono proseguire. La soluzione sta nell'introduzione di **condition variables**, con due operazioni associate: *wait* e *signal*. Quando una procedura scopre di non poter continuare chiama una wait su una qualche *variabile di tipo condizione* (nell'esempio potrebbe essere la variabile *pieni*). Un altro processo invece, quando vuole svegliare il partner sospeso, chiama una signal sulla stessa variabile.

Un ultima cosa da fare, per evitare che due processi siano nel monitor contemporaneamente dopo la signal, è definire cosa succede. Una scelta possibile è che chi chiama la signal debba istantaneamente uscire dal monitor.

Se viene eseguita una signal su una variabile su cui nessuno è sospeso, il segnale viene perso.

6.7.1 Produttore-Consumatore con i monitor Java

In Java, aggiungendo la parola chiave **synchronized** alla dichiarazione di un metodo, si garantisce che una volta che un thread ha iniziato l'esecuzione di quel metodo, a nessun altro thread sarà permesso di eseguire un metodo synchronized di quella stessa classe. Una soluzione al problema produttore-consumatore utilizzando i monitor in Java è mostrata in figura 6.5.

```
public class ProduttoreConsumatore {
      static finale int N = 100; // Costante che dà la dimensione del buffer
      static produttore p = new produttore( ); // Istanzia un nuovo thread produttore
      static consumatore c = new consumatore( ); // Istanzia un nuovo thread consumatore
      static nostro_monitor mon = new nostro_monitor( ); // Istanzia un nuovo monitor

      public static void main(String args[ ]) {
         p.start( ); // Inizia il thread produttore
         c.start( ); // Inizia il thread consumatore
      }

      static class produttore extends Thread {
         public void run( ) { // Esegue un metodo che contiene codice di thread
            int item;
            while (true) { // Ciclo produttore
               item = produci_elemento( );
               mon.inserisci(item);
            }
         }
         private int produci_elemento( ) { ... } // Produce veramente
      }

      static class consumatore extends Thread {
         public void run( ) { // Esegue un metodo che contiene codice di thread
            int item;
            while (true) { // Ciclo consumatore
               item = mon.estrai( );
               consuma_elemento (item);
            }
         }
         private void consuma_elemento(int item) { ... } // Consuma veramente
      }

      static class nostro_monitor { // Questo è un monitor
         private int buffer[ ] = new int[N];
         private int cont = 0, lo = 0, hi = 0; // Contatori e indici

         public synchronized void inserisci(int val) {
            if (cont == N) vai_a_dormire( ); // Se il buffer è pieno, sospenditi
            buffer [hi] = val; // Inserisce un elemento nel buffer
            hi = (hi + 1) % N; // Posizione in cui mettere il prossimo elemento
            cont = cont + 1; // C'è un nuovo elemento nel buffer, ora
            if (cont == 1) notify( ); // Se il consumatore è sospeso, sveglialo
         }

         public synchronized int estrai( ) {
            int val;
            if (cont == 0) vai_a_dormire( ); // Se il buffer è vuoto, sospenditi
            val = buffer [lo]; // Prendi un elemento dal buffer
            lo = (lo + 1) % N; // Posizione da cui prendere elementi dal buffer
            cont = cont - 1; // Un elemento in meno nel buffer
            if (cont == N - 1) notify( ); // Se il produttore è sospeso, sveglialo
            return val;
         }
         private void vai_a_dormire( ) { try{wait( );} catch(InterruptedException exc) {};}
      }
}
```

Figure 6.5: Produttore-Consumatore con monitor.

I metodi sincronizzati di Java differiscono in modo sostanziale dai monitor tradizionali: Java non ha variabili di tipo condizione, piuttosto due procedure, *wait* e *notify*, equivalenti di *sleep* e *wake up*, eccetto che, quando usate da metodi sincronizzati, non sono soggette a corse critiche.

6.8 Lo scambio di messaggi

La differenza tra le soluzioni proposte è che i semafori sono troppo a basso livello e di difficile gestione, e i monitor non sono utilizzabili in molti linguaggi di programmazione. Inoltre entrambe le soluzioni non permettono lo scambio di informazioni tra macchine. Per quello è necessario qualcosa di diverso: **lo scambio di messaggi**.

Questo metodo di comunicazione tra processi usa due primitive:

```
send(dest, &message);
receive(source, &message);
```

Esse sono chiamate di sistema, non costrutti del linguaggio, possono essere quindi inserite in procedure di libreria.

La prima spedisce un messaggio ad una destinazione. La seconda lo riceve da una determinata sorgente, o da *ANY*. Se non è disponibile alcun messaggio il ricevente potrebbe bloccarsi fino all'arrivo di uno.

6.8.1 Problematiche di progetto dei sistemi

Se le macchine che comunicano stanno sulla rete i messaggi possono essere persi. Per prevenire che ciò sia fatale si potrebbe pensare ad un messaggio di **acknowledgement**, il quale, se non ricevuto in un tempo limite dal mittente, suggerisce a quest'ultimo il re-invio del messaggio.

Un altro problema riguarda l'**autenticazione**, altre volte sono un problema le prestazioni... insomma, questi sistemi sono soggetti a parecchie vulnerabilità a cui è necessario prestare attenzione.

6.8.2 Produttore-Consumatore con scambio di messaggi

Vediamo come può essere risolto il problema con scambio di messaggi e senza memoria condivisa. Una soluzione è in figura 6.6

```
#define N 100                                   / * Numero di posizioni nel buffer * /

void Produttore(void)
{
   int elemento;
   messaggio m;                                 / * Buffer per i messaggi * /

   while (TRUE) {
      elemento = produci_elemento( );           / * Genera qualcosa da mettere nel buffer * /
      receive(Consumatore, &m);                 / * Aspetta che arrivi un messaggio vuoto * /
      costruisci_messaggio(&m, elemento);       / * Costruisci un messaggio da spedire * /
      send(Consumatore, &m);                    / * Spedisci un elemento al consumatore * /
   }
}

void Consumatore(void)
{
   int elemento, i;
   messaggio m;

   for (i = 0; i < N; i++) send(Produttore, &m); / * Spedisci N messaggi vuoti * /
   while (TRUE) {
      receive(Produttore, &m);                  / * Prendi un messaggio un elemento * /
      elemento = estrai_elemento(&m);           / * Estrai un elemento dal messaggio * /
      send(Produttore, &m);                     / * Rimanda indietro una risposta vuota * /
      consuma_elemento(elemento);               / * Fai qualcosa con l'elemento * /
   }
}
```

Figure 6.6: Produttore-Consumatore con N messaggi.

Un modo di indirizzare i messaggi è quello di assegnare ad ogni processo un indirizzo unico e di inviargli direttamente i messaggi. Un altro prevede di usare una nuova struttura dati, detta **mailbox**, che bufferizza uno specifico numero di messaggi. Nel secondo caso i parametri di *send* e *receive* sono mailbox e non processi.

Quando un processo tenta di mandare un messaggio ad una mailbox piena esso viene sospeso finché non viene fatto spazio per il suo messaggio.

Un ultimo modo è il **rendez-vous**, esso è meno flessibile ma più facile da implementare: viene eliminata completamente la bufferizzazione, se la *send* viene chiamata prima della *receive* il mittente viene bloccato fino al sopraggiungere di quest'ultima; analogamente se la *receive* è chiamata prima della *send* è il destinatario a rimanere bloccato fino all'invio del messaggio.

6.9 Barriere

Un meccanismo di sincronizzazione per gruppi di processi. Alcune applicazioni sono divise in fasi e hanno la regola che nessun processo può proseguire finché tutti sono pronti per la fase successiva. Questo comportamento può essere ottenuto mediante una **barriera** alla fine di ogni fase: quando un processo la raggiunge viene bloccato finché tutti la raggiungono.

Un esempio è mostrato in figura 6.7.

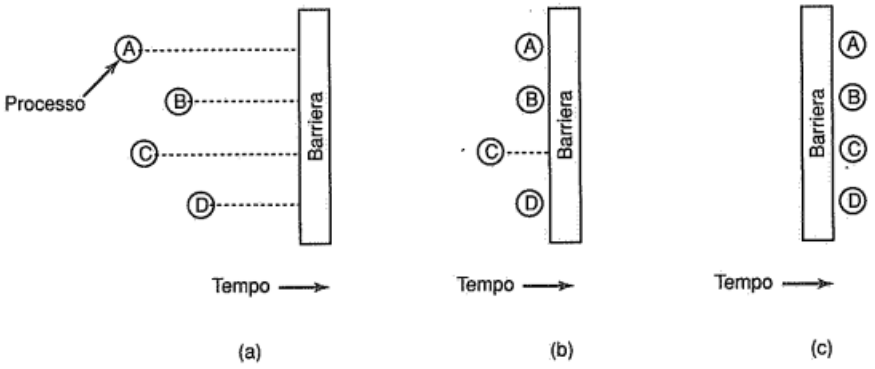


Figure 6.7: Uso di una barriera.

Il primo processo finisce i calcoli e chiama la primitiva *barrier*, generalmente con una procedura di libreria, dopodiché viene sospeso. Quando anche l'ultimo processo chiama la primitiva tutti vengono liberati e si procede.

6.9.1 CyclicBarrier di Java

...

6.9.2 CountDownLatch di Java

...

6.10 Problemi classici di IPC

6.10.1 Filosofi a cena

"Cinque filosofi sono seduti attorno ad un tavolo rotondo, ciascuno ha un piatto di spaghetti e due forchette, posizionate a destra e sinistra rispetto al piatto; per mangiare il filosofo ha bisogno di entrambe le forchette; la vita dei filosofi si divide in periodi un cui pensano e periodi in cui mangiano. Quando un filosofo ha fame prende le forchette, mangia, ed in seguito depone le forchette e torna a pensare."

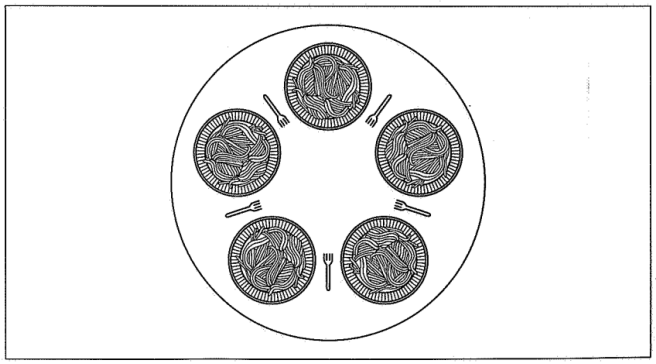


Figure 6.8: L'ora di cena al dipartimento di filosofia.

Occorre scrivere un programma che rappresenti tutti i filosofi, e che non si fermi mai.

Quello presentato è il famoso problema dei filosofi a cena. È utile per modellare processi che competono pr l'accesso esclusivo ad un numero limitato di risorse (come i dispositivi in/out).

La soluzione banale non funziona, poiché se tutti i filosofi prendono contemporaneamente la forchetta non ne rimangono abbastanza e l'intero programma si blocca per sempre, senza fare progressi (una situazione di questo tipo è detta **starvation**).

Una popolare soluzione prevede di far attendere un piccolo tempo casuale al filosofo prima di ritentare a prendere le forchette. Tuttavia esistono situazioni estremamente delicate in cui non è possibile affidarsi a dei numeri casuali.

Una prima soluzione, ancora non perfetta, prevede l'utilizzo di un semaforo binario: prima di prendere possesso delle forchette il filosofo chiama una *down* su un mutex, dopo averle deposte invece ci chiama una *up*. Questa soluzione non è ottimale in quanto è un solo filosofo alla volta a mangiare, se fosse ottimale con 5 forchette sono due i filosofi a poter mangiare in contemporanea.

Una soluzione definitiva (Figura 6.9) usa un vettore *affamato*, per tenere traccia se un filosofo sta mangiando, pensando o aspettando. In questo caso un filosofo si può portare nello stato in cui mangia solo se nessuno dei suoi vicini sta mangiando. Il programma usa un vettore di semafori, uno per filosofo, in modo tale da permettere ai filosofi di bloccarsi nel caso in cui debbano aspettare.

```
#define N              5            / * Numero di filosofi * /
#define SINISTRA       (i+ N - 1)%N / * Numero vicino di sinistra di i * /
#define DESTRA         (i+1)%N      / * Numero vicino di destra di i * /
#define PENSANTE       0            / * Il filosofo sta pensando * /
#define AFFAMATO       1            / * Il filosofo cerca di prendere le forchette *
#define MANGIANTE      2            / * Il filosofo sta mangiando * /
typedef int semaforo;               / * I semafori sono un tipo di intero * /
int stato[N];                       / * Vettore per tenere traccia dello stato di ognuno * /
semaforo mutex = 1;                 / * Mutua esclusione regioni critiche * /
semaforo s[N];                      / * Un semaforo per filosofo * /

void filosofo(int i)                / * i: numero filosofo, da 0 a N - 1 * /
{
      while (TRUE) {                / * Ripeti per sempre * /
         pensa( );                  / * Il filosofo sta pensando * /
         prendi_forchette(i);       / * Ottiene due forchette si blocca * /
         mangia( );                 / * Yum-yum, spaghetti * /
         posa_forchette(i);         / * Rimette le forchette sul tavolo * /
      }
}

void prendi_forchette(int i)        / * i: numero filosofo, da 0 a N - 1 * /
{
      down(&mutex);                 / * Entra nella regione critica * /
      stato[i] = AFFAMATO;          / * Registra che il filosofo i è affamato * /
      test(i);                      / * Cerca di ottenere due forchette * /
      up(&mutex);                   / * Esce dalla regione critica * /
      down(&s[i]);                  / * Si blocca se forchette non ottenute * /
}

void posa_forchette(i)              / * i: numero filosofo , da 0 a N - 1 * /
{
      down(&mutex);                 / * Entra nella regione critica * /
      stato[i] = PENSANTE;          / * Il filosofo ha finito di mangiare * /
      test(SINISTRA);               / * Vede se il vicino di sinistra può mangiare * /
      test(DESTRA);                 / * Vede se il vicino di destra può mangiare * /
      up(&mutex);                   / * Esce dalla regione critica * /
}
void test(i)                        / * i: numero filosofo , da 0 a N - 1 * /
{
      if (stato[i] == AFFAMATO && stato[SINISTRA] != MANGIANTE && stato[DESTRA] != MANGIANTE)
      {
          stato[i] = MANGIANTE;
          up(&s[i]);
      }
}
```

Figure 6.9: Una soluzione ottimale al problema dei filosofi.

6.10.2 Lettori e scrittori

Un altro problema noto è quello dei lettori e scrittori, che modella il problema di accesso ad una base di dati. Supponendo di avere n processi che tentano di accedere ad un database dobbiamo fare in modo che sia i processi che scrivono che quelli che leggono possano competere in modo sano per la risorsa.

Una soluzione è mostrata in Figura 6.10. In questo caso il primo lettore che ottiene l'accesso esegue una *down* sul semaforo *db*. I successivi lettori incrementano solamente un contatore, *rc*, l'ultimo lettore che se ne va invece chiama la *up*, permettendo all'eventuale scrittore bloccato di accedere.

Questa strategia va KO se arrivano spesso lettori, è possibile che lo scrittore aspetti per sempre. Per ovviare si può fare in modo che un nuovo lettore che arriva si accodi agli scrittori in attesa, tuttavia anche questa soluzione presenta problemi di prestazioni. Esistono anche soluzioni che favoriscono gli scrittori.

```
typedef int semaforo;                    / * Indovina? * /
semaforo mutex = 1;                      / * Controlla gli accessi a 'rc' * /
semaforo db = 1;                         / * Controlla accessi alla base dati * /
int rc = 0;                              / * Numero processi che leggono o vogliono leggere * /

void lettore(void)
{
       while (TRUE) {                    / * Ripeti all'infinito * /
           down(&mutex);                 / * Ottieni accesso esclusivo a 'rc' * /
           rc = rc + 1;                  / * Un lettore in più ora * /
           if (rc == 1) down(&db);       / * Se questo è il primo lettore ...* /
           up(&mutex);                   / * Rilascia accesso esclusivo a 'rc' * /
           leggi_base_dati( );           / * Accedi ai dati * /
           down(&mutex);                 / * Ottieni accesso esclusivo a 'rc' * /
           rc = rc - 1;                  / * Un lettore in meno, ora * /
           if (rc == 0) up(&db);         / * Se questo è l'ultimo lettore ...* /
           up(&mutex);                   / * Rilascia accesso esclusivo ad 'rc' * /
           leggi_dati_utente( );         / * Sezione non critica * /
       }
}

void scrittore(void)
{
       while (TRUE) {                    / * Ripeti all'infinito * /
           pensa_ai_dati( );             / * Sezione non critica * /
           down(&db);                    / * Ottieni l'accesso esclusivo * /
           scrivi_base_dati( );          / * Aggiorna i dati * /
           up(&db);                      / * Rilascia l'accesso esclusivo * /
       }
}
```

Figure 6.10: Una soluzione per il problema lettori e scrittori.

6.10.3 Il barbiere che dorme (facoltativo)

"Nel negozio c'è un barbiere, una sedia da lavoro e n sedie per far sedere i clienti in attesa. Se non ci sono clienti il barbiere si siede sulla sedia da lavoro e dorme. Quando arriva un cliente sveglia il barbiere. Se arrivano altri clienti mentre sta lavorando si siedono ad aspettare sulle n sedie, se sono tutte occupate il nuovo cliente se ne va."

il problema in questo caso è programmare il barbiere ed i clienti senza arrivare ad avere corse critiche.

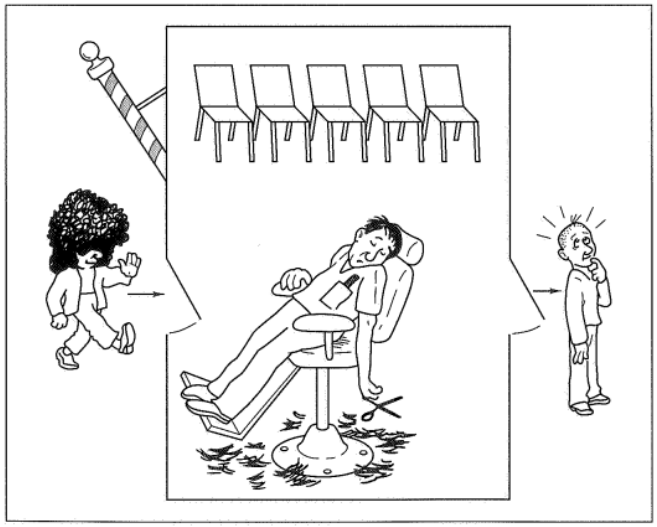


Figure 6.11: Un tipico barbiere del sud.

Una soluzione possibile usa tre semafori: *clienti* che conta il numero dei clienti in attesa sulle sedie, *barbieri* che conta il numero di barbieri (0 o 1) inattivi, e *mutex* che viene usato per la mutua esclusione. Viene aggiunta una variabile copia di *clienti* che permette ai nuovi clienti di leggere quanti sono ad aspettare (senza questa nuova variabile non ci sarebbe stato modo di leggere il semaforo).

La mattina il barbiere esegue la procedura *barbiere*, che lo blocca sul semaforo clienti, va quindi a dormire finché non arriva il primo cliente. Quando quest'ultimo arriva esegue la procedura *cliente*, iniziando con l'acquisizione di *mutex* per entrare in regione critica. In seguito controlla se il numero di clienti in attesa non riempie le sedie e, in caso negativo, rilascia *mutex* e se ne va senza taglio di capelli. Se invece c'è posto incrementa la variabile *in_attesa*, esegue una *up* su clienti, risvegliando il barbiere. Quando il cliente rilascia *mutex* il barbiere gli taglia i capelli.

```
#define SEDIE 5                     / * Numero sedie per clienti in attesa * /

typedef int semaforo;               / * Indovina?* /

semaforo clienti = 0;               / * Numero clienti in attesa  * /
semaforo barbieri = 0;              / * Numero barbieri in attesa di clienti * /
semaforo mutex = 1;                 / * Per la mutua esclusione * /
int in_attesa = 0;                  / * Clienti in attesa (non durante il taglio) * /

void barbiere(void)
{
      while (TRUE) {
         down(&clienti);            / * Sospenditi se numero clienti zero * /
         down(&mutex);              / * Acquisisci accesso a in_attesa * /
         in_attesa = in_attesa - 1; / * Decrementa contatore clienti in attesa * /
         up(&barbieri);             / * Un barbiere è pronto per il taglio * /
         up(&mutex);                / * Rilascia in_attesa * /
         taglia_capelli( );         / * Taglia capelli (fuori sezione critica) * /
      }
}

void cliente(void)
{
      down(&mutex);                 / * Entra nella regione critica * /
      if (in_attesa < CHAIRS) {     / * Se non ci sono sedie libere, vattene * /
         in_attesa = in_attesa + 1; / * Incrementa numero di clienti in attesa * /
         up(&clienti);              / * Sveglia un barbiere, se necessario * /
         up(&mutex);                / * Rilascia l'accesso a in_attesa * /
         down(&barbieri);           / * Sospenditi se numero barbieri liberi zero * /
         vai_al_taglio( );          / * Siediti per essere servito * /
      } else {
         up(&mutex);                / * Il negozio è pieno; non attendere * /
      }
}
```

Figure 6.12: Una soluzione per il problema del barbiere.

Sincronizzazione in Java

...

Pipe