

SISTEMI OPERATIVI

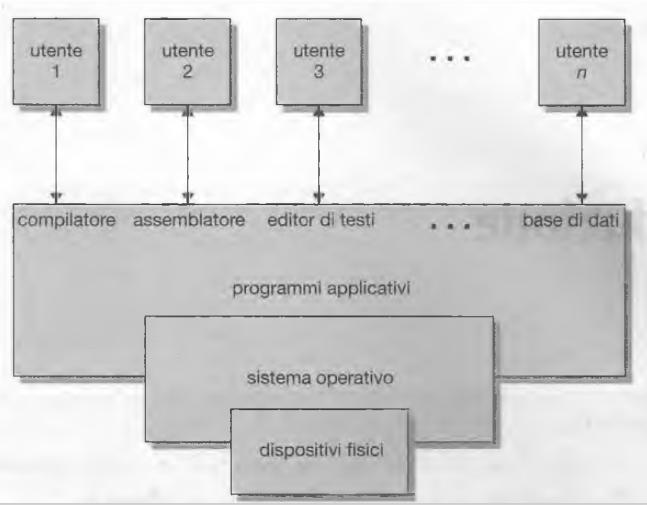
PRIMO CAPITOLO:

Che cos'è un sistema operativo:

Sistema Operativo(SO)=programma che agisce come *intermediario* tra utente e elementi fisici del PC. Lo scopo è fornire un ambiente *conveniente* ed *efficiente*.

Il SO dal punto di vista dell'utente—> si considera principalmente la *facilità d'uso* ma senza dare attenzione *all'utilizzo delle risorse*, a come cioè sono condivise le risorse hardware e software, quindi si focalizzano di più sull'esperienza del singolo utente più che sull'utilizzo delle risorse.

Il SO dal punto di vista del sistema/calcolatore—> si considera come un *assegnatore di risorse*, il SO deve decidere come assegnarle agli specifici programmi e utenti affinché *il sistema di calcolo*(dispositivi fisici=>(CPU; memoria; dispositivi di input e output; I/O), sistema operativo =>(controlla e coordina l'uso dei dispositivi da parte dei programmi applicativi per gli utenti), programmi applicativi =>(editor di testo, fogli di calcolo, compilatori e programmi di consultazione del Web) e utenti) *operi in modo equo ed efficiente* .



L'assegnazione delle risorse è importante soprattutto nel caso in cui molti utenti accedono agli stessi mainframe o minicalcolatori.SO è anche un *programma di controllo* soprattutto per il funzionamento e il controllo dei dispositivi di I/O.

Un'altra definizione del SO = è il solo programma che funziona sempre nel calcolatore, generalmente chiamato **KERNEL**(nucleo).

Funzionamento di un sistema di calcolo:

L'avviamento del sistema richiede un programma d'avviamento ,la sua funzione consiste nell'inizializzare i diversi componenti del sistema, dai registri della CPU ai controllori dei diversi dispositivi, fino al contenuto della memoria centrale.Deve caricare nella memoria il SO e avviare l'esecuzione, perciò individua e carica nella memoria il kernel del SO;

il SO avvia quindi l'esecuzione del primo processo d'elaborazione e attende che si verifichi qualche evento.Un evento è segnalato da **un'interruzione** dell'attuale sequenza d'esecuzione della CPU, che può essere causata da un dispositivo fisico —>(INTERRUPT =interruzione), segnali inviati alla CPU attraverso il bus di sistema o da un programma -> (EXCEPTION o TRAP=eccezione) a seguito di una richiesta specifica effettuata da un programma utente per ottenere l'esecuzione di un servizio del SO, attraverso un'istruzione detta SYSTEM CALL (chiamata di sistema) o SUPERVISOR CALL(chiamata supervisore).

La **gestione di un'interruzione** deve essere molto rapida perciò, considerando che il numero dei possibili segnali d'interruzione è predefinito, si può usare una tabella di puntatori alle specifiche procedure.L'accesso a questa sequenza d'indirizzi, detta **vettore delle interruzioni**, avviene per mezzo di un indice, codificato nello stesso segnale d'interruzione ed ha lo scopo di fornire l'indirizzo di servizio relativo all'interruzione ed inoltre salvare l'indirizzo dell'istruzione interrotta nella pila (stack) di sistema.

Una volta terminato il servizio dell'interruzione, l'indirizzo di ritorno precedentemente salvato viene caricato nel **contatore di programma** (*program counter*), che contiene l'indirizzo della prossima istruzione da eseguire, consentendo la ripresa della computazione interrotta come se nulla fosse accaduto.

Struttura della memoria :

La CPU può caricare istruzioni esclusivamente dalla memoria, la memoria principale è chiamata anche **memoria ad accesso diretto** (*random access memory, RAM*). Tutte le tipologie di memoria forniscono un vettore di parole. Ciascuna parola possiede un proprio indirizzo. L'interazione avviene per mezzo di una sequenza di istruzioni **load** (trasferisce il contenuto di una parola della memoria centrale in uno dei registri interni della CPU)e **store** (copia il contenuto di uno di questi registri nella locazione di memoria specificata) opportunamente indirizzate.

Oltre a questi accessi la CPU preleva automaticamente dalla RAM le istruzioni da eseguire. In teoria, si vorrebbe che sia i programmi sia i dati da essi trattati potessero risiedere in modo permanente nella memoria centrale. Questo non è possibile per i seguenti due motivi:

1.-la capacità della memoria centrale non è di solito sufficiente a contenere in modo permanente tutti i programmi e i dati richiesti;

2.-la memoria centrale è un dispositivo di memorizzazione *volatile*, sicché perde il proprio contenuto quando si spegne il sistema o si ha un'interruzione dell'alimentazione elettrica.

Oltre a caratterizzarsi per velocità e costo, i sistemi di memorizzazione si suddividono in **volatili e non volatili**; la **memoria volatile** comporta la perdita dei dati nel caso di interruzione dell'alimentazione. Qualora non si disponga di sistemi di salvataggio automatico dei dati (*sistemi di backup*) è necessario salvare i dati su **memoria non volatile**.

Sistemi monoprocessoresso :

Un sistema monoprocessoresso è dotato di una CPU principale in grado di eseguire un insieme di istruzioni. L'utilizzo di microprocessori con finalità specifiche è comune e non trasforma un sistema monoprocessoresso in un sistema multiprocessoresso. Se è presente una sola **CPU**, si tratta di un sistema monoprocessoresso.

Sistemi multiprocessoresso :

Questo tipo di sistemi dispone di più unità d'elaborazione in stretta comunicazione, che condividono i canali di comunicazione all'interno del calcolatore (*bus*), i timer dei cicli di macchina (*clock*) e talvolta i dispositivi di memorizzazione e periferici.

Tali sistemi hanno tre vantaggi principali:

1. **Maggiore produttività (throughput)**. Aumentando il numero di unità d'elaborazione è possibile svolgere un lavoro maggiore in meno tempo.
2. **Economia di scala**. Se più programmi devono operare sullo stesso insieme di dati, è economicamente più conveniente registrarli in dischi condivisi da tutte le unità d'elaborazione, piuttosto che avere più calcolatori con i rispettivi dischi locali e più copie degli stessi dati.
3. **Incremento dell'affidabilità**. Un guasto di alcune di loro non blocca il sistema, semplicemente lo rallenta; ciascuna delle unità d'elaborazione rimanenti assume su di sé una parte del lavoro che svolgevano le unità d'elaborazione guaste; l'intero sistema non si ferma, ma funziona a una velocità ridotta.

I sistemi multiprocessoresso attualmente in uso sono di due tipi.

Alcuni impiegano la **multielaborazione asimmetrica** (*asymmetric multiprocessing*, AMP), in cui a ogni unità d'elaborazione si assegna un compito specifico. —> piano di gerarchia

Nei sistemi più comuni si ricorre alla **multielaborazione simmetrica** (*symmetric multiprocessing*, SMP), in cui ogni processore è abilitato al compimento di tutte le operazioni del sistema. —> piano di parità. Il vantaggio offerto da questo modello è che molti processi sono eseguibili contemporaneamente (*n* processi se si hanno *n* CPU) senza causare un rilevante calo delle prestazioni.

Una delle ultime innovazioni, i cosiddetti **server blade**, che accolgono nello stesso contenitore fisico le schede del processore, dell'I/O e della rete. A differenza dei tradizionali sistemi multiprocessoresso, nei server blade ogni scheda madre (una scheda, cioè, che ospita una **CPU**) avvia ed esegue in maniera indipendente il proprio sistema operativo.

Struttura del sistema operativo:

Il sistema operativo costituisce l'ambiente esecutivo dei programmi. Fra le più importanti caratteristiche dei sistemi operativi vi è la multiprogrammazione che consente di aumentare la percentuale d'utilizzo della CPU, organizzando i lavori in modo tale da mantenerla in continua attività. Il sistema operativo tiene contemporaneamente in memoria centrale diversi lavori. Dato che, in genere, la memoria centrale è troppo piccola per contenere tutti i programmi da eseguire, questi vengono collocati inizialmente sul disco in un'area apposita, detta **job pool**, contenente tutti i processi in attesa di essere allocati nella memoria centrale. Il sistema operativo ne sceglie uno tra quelli contenuti nella memoria e inizia l'esecuzione: a un certo punto potrebbe trovarsi nell'attesa di qualche evento, come il completamento di un'operazione di I/O. In questi casi, in un sistema non multiprogrammato, la CPU rimarrebbe inattiva. In un sistema con multiprogrammazione, invece, il sistema operativo passa semplicemente a un altro lavoro e lo esegue. Quando il primo lavoro ha terminato l'attesa, la CPU ne riprende l'esecuzione. Finché c'è almeno un lavoro da eseguire, la CPU non è mai inattiva. Esempio = l'avvocato non segue un solo cliente alla volta. Un sistema con multiprogrammazione fornisce un ambiente in cui le risorse del sistema (per esempio **CPU**,

memoria, dispositivi periferici) sono impiegate in modo efficiente, ma non rappresenta un sistema d'interazione con l'utente. La **partizione del tempo d'elaborazione** (*time sharing* o *multitasking*) è un'estensione logica della multiprogrammazione. Un programma caricato in memoria e predisposto per la fase d'esecuzione è noto come **processo**.

La partizione del tempo di elaborazione e la multiprogrammazione richiedono la contemporanea presenza di diversi processi in memoria. Se alcuni processi sono pronti per il trasferimento in memoria centrale, ma lo spazio disponibile non è sufficiente per accogliere tutti, il sistema deve fare una selezione; questa scelta, illustrata nel Capitolo 5, si chiama **job scheduling**, ossia *pianificazione dei lavori*. In un sistema basato sulla partizione del tempo di elaborazione, il sistema operativo deve garantire tempi di risposta accettabili: questa finalità è raggiunta, in alcuni casi, grazie alla tecnica detta **swapping** (*avvicendamento*), che consente di scambiare i processi presenti in memoria con quelli che risiedono su disco e viceversa. Un metodo più comune per ottenere il medesimo risultato è la **memoria virtuale**, tecnica che consente l'esecuzione di lavori d'elaborazione anche non interamente caricati nella memoria. Il più evidente vantaggio della memoria virtuale è che i programmi possono avere dimensioni maggiori della **memoria fisica**; inoltre, essa astrae la memoria centrale in un grande e uniforme vettore, separando la **memoria logica**, vista dall'utente, dalla memoria fisica, sollevando i programmatori dai problemi legati ai limiti della memoria.

Attività del sistema operativo—>Dupliche modalità di funzionamento:

Sono necessarie almeno due diverse modalità: **modalità utente** e **modalità di sistema** detta anche modalità kernel, modalità supervisore, modalità monitor o modalità privilegiata). Per indicare quale sia la modalità attivo, l'architettura della **CPU** deve essere dotata di un bit, chiamato appunto **bit di modalità**: di sistema (0) o utente (1). Questo bit consente di stabilire se l'istruzione corrente si esegue per conto del sistema operativo o per conto di un utente. All'avviamento del sistema, il bit è posto in modalità di sistema. Si carica il sistema operativo che provvede all'esecuzione dei processi utenti in modalità utente. Ogni volta che si verifica un'interruzione o un'eccezione si passa dalla modalità utente a quella di sistema, cioè si pone a 0 il bit di modo. Perciò quando il sistema operativo riprende il controllo del calcolatore si trova in modalità di sistema. Prima di passare il controllo al programma utente, il sistema ripristina la modalità utente riportando a 1 il valore del bit.

La duplice modalità di funzionamento (*dual mode*) consente la protezione del sistema: operativo rispetto al comportamento degli utenti e viceversa. Questo livello di protezione si ottiene definendo come **istruzioni privilegiate** le istruzioni di macchina che possono causare danni allo stato del sistema. Poiché la **CPU** consente l'esecuzione di queste istruzioni soltanto nella modalità di sistema, se si tenta di far eseguire in modalità utente un'istruzione privilegiata, la **CPU** non la esegue, ma la tratta come un'istruzione illegale inviando un segnale di eccezione al sistema operativo.

Le chiamate di sistema sono gli strumenti con cui un programma utente richiede al sistema operativo di compiere operazioni a esso riservate, per conto del programma utente. In ogni caso, però, le chiamate di sistema sono il mezzo utilizzato dai processi per sollecitare all'azione il sistema operativo. Una chiamata di sistema è solitamente realizzata come un'eccezione che rimanda a un indirizzo specifico nel vettore delle interruzioni. Quando un programma utente esegue una chiamata di sistema, questa è gestita dalla **CPU** come un'interruzione. Il controllo passa, tramite il vettore delle interruzioni, all'apposita procedura di servizio presente all'interno del sistema operativo e si pone il bit di modo in modalità di sistema. Se la **CPU** non dispone di queste due modalità di funzionamento il sistema operativo può andare incontro a serie limitazioni. Un programma utente potrebbe cancellare il sistema operativo scrivendo nuove informazioni nelle locazioni in cui questo è memorizzato, e più programmi potrebbero scrivere contemporaneamente nello stesso dispositivo, con l'eventualità di ottenere risultati disastrosi.

Timer

Occorre assicurare che il sistema operativo mantenga il controllo dell'elaborazione, cioè impedire che un programma utente entri in un ciclo infinito o non richieda servizi del sistema senza più restituire il controllo al sistema operativo. A tale scopo si può usare un timer, programmabile affinché invii un segnale d'interruzione alla **CPU** a intervalli di tempo specificati. La presenza di un timer garantisce quindi che nessun programma utente possa essere eseguito troppo a lungo. Una tecnica semplice consiste nell'impostare un contatore con un valore pari al tempo concesso al programma per la propria esecuzione. Per esempio, se il programma richiede 7 minuti si

dovrebbe impostare il contatore al valore 420. Il timer genererà un'interruzione ogni secondo e il contatore sarà decrementato di 1; fintanto che il valore resta positivo, il controllo ritorna al programma utente; quando il contatore raggiunge valore negativo, il sistema operativo termina l'esecuzione del programma per il superamento del tempo a esso assegnato.

Gestione dei processi

Processo= programma in esecuzione, un lavoro d'elaborazione, programma eseguito in un ambiente a partizione del tempo d'elaborazione.

Un processo ha bisogno di tempo di CPU, memoria, file e dispositivi I/O.

Il programma di per sé NON è UN PROCESSO MA UN'ENTITÀ PASSIVA, mentre il processo è un entità ATTIVA con un contatore di programma che indica la successiva istruzione da eseguire (Thread). L'esecuzione di un processo deve essere sequenziale: la CPU esegue le istruzioni del processo una dopo l'altra fino al termine e in ogni istante si esegue al massimo un'istruzione del processo. Quindi anche se due processi possono corrispondere allo stesso programma, si considerano in ogni caso due sequenze d'esecuzione separate .

Gestione della memoria

La CPU fa riferimento sempre solo alla memoria centrale e genera indirizzi assoluti.

• Gestione della memoria di massa

- il sistema operativo fornisce un'interfaccia logica uniforme per la memorizzazione delle informazioni definendo un'unità logica di archiviazione, il FILE.
- Un file è una raccolta d'informazioni correlate definite dal loro creatore. Comune mente, i file rappresentano programmi, sia sorgente sia oggetto, e dati. I file di dati possono essere numerici, alfabetici, alfanumerici o binari; la loro forma può essere libera, come nei file di testo, oppure rigidamente formattata, per esempio in campi fissi. Il concetto di file è chiaramente molto generale. Giacché la memoria centrale è troppo piccola per contenere tutti i dati e tutti i programmi, e il suo contenuto va perduto se il sistema si spegne, il calcolatore deve disporre di una memoria secondaria a sostegno della memoria centrale. La maggior parte dei moderni sistemi di calcolo impiega i dischi come principale mezzo di memorizzazione secondaria, sia per i programmi sia per i dati. I dischi contengono la maggior parte dei programmi, compresi i compilatori, gli assemblatori, le procedure di ordinamento e gli editor. Il frequente uso della memoria secondaria impone una sua gestione efficiente. Infatti, l'efficienza complessiva di un calcolatore può dipendere dalla velocità del sottosistema di gestione dei dischi e dagli algoritmi che lo gestiscono.

Cache

Di norma le informazioni sono mantenute in unità di memoria come la memoria centrale; al momento del loro uso si copiano temporaneamente in un'unità più veloce: la cache. Quando si deve accedere a una particolare informazione, innanzitutto si controlla se è già presente all'interno della cache; in tal caso si adopera direttamente la copia contenuta nella cache, altrimenti si preleva dalla memoria centrale e si copia nella cache, poiché si suppone che questa informazione presto servirà ancora. Inoltre, i registri programmabili presenti all'interno della CPU, come i registri indice, rappresentano per la memoria centrale una cache ad alta velocità.

Sommario 1° CAPITOLO

Un sistema operativo è il software che gestisce l'hardware di un calcolatore, e fornisce un ambiente all'interno del quale siano eseguibili le applicazioni. Forse l'aspetto più concreto dei sistemi operativi è l'interfaccia d'accesso al computer che essi forniscono all'utente.

Per essere eseguiti, i programmi devono risiedere nella memoria centrale del calcolatore. Questa è infatti la sola area di memoria di grandi dimensioni direttamente accessibile dalla CPU. Essa è un vettore, di parole o byte, di dimensioni variabili tra i milioni e i miliardi di byte. Ciascuna parola possiede il proprio indirizzo. La memoria centrale è un dispositivo volatile, poiché perde il proprio contenuto quando manca l'alimentazione elettrica. La maggior parte dei sistemi di calcolo dispone di una memoria secondaria come estensione della memoria centrale. Il requisito fondamentale della memoria secondaria è la capacità di memorizzare in modo permanente grandi quantità di dati. Il più comune dispositivo di memoria secondaria è il disco magnetico; si tratta di un dispositivo di memoria non volatile che può memorizzare sia dati sia programmi e che consente l'accesso diretto ai suoi elementi.

I sistemi di memorizzazione di un calcolatore si possono organizzare in modo gerarchico secondo la velocità e il costo. I livelli più alti rappresentano i dispositivi più rapidi, ma più costosi. Scendendo nella gerarchia, il costo per bit generalmente decresce, mentre di solito aumentano i tempi d'accesso.

Alla base della progettazione di un sistema di elaborazione sono possibili approcci di versi. Una prima scelta deve essere operata tra i sistemi a processore unico e quelli con due o più processori; in quest'ultimo caso, la memoria fisica e i dispositivi periferici sono condivisi da tutti i processori. Lo schema più comune tra i sistemi multiprocessore è rappresentato dalla multielaborazione simmetrica (SMP), che vede tutti i processori su un piano di parità: essi elaborano in piena indipendenza gli uni dagli altri. Una forma specializzata di multielaborazione è invece rappresentata dai cluster di elaboratori (*clustered systems*), ovvero un certo numero di elaboratori connessi da una rete locale. Per utilizzare al meglio la **CPU**, i sistemi operativi moderni impiegano la multiprogrammazione, grazie alla quale diversi processi possono occupare contemporaneamente la memoria, assicurando che la **CPU** non resti mai inattiva. Con i sistemi a tempo ripartito il concetto di multiprogrammazione è stato ulteriormente esteso, per mezzo di algoritmi di scheduling della **CPU** che fanno la spola tra un processo e l'altro, dando così l'impressione che molti processi siano in esecuzione allo stesso tempo.

Il sistema operativo deve assicurare il corretto funzionamento del calcolatore. Per evitare che i programmi utenti interferiscano tra di loro e col sistema operativo, la **CPU** ha due modalità di funzionamento: la modalità utente e la modalità di sistema. Diverse istruzioni (come quelle di **I/O** e di arresto del calcolatore) sono privilegiate e si possono eseguire sola mente in modalità di sistema. Anche l'area della memoria in cui risiede il sistema operativo deve essere protetta dai tentativi di modifiche da parte degli utenti. La presenza di un timer evita il verificarsi di cicli infiniti. Queste funzioni (duplice modalità di funzionamento, istruzioni privilegiate, protezione della memoria, interruzioni del timer) costituiscono gli elementi fondamentali impiegati dal sistema operativo per ottenere un corretto funzionamento del sistema.

Un processo (*process* o *job*) è l'unità fondamentale di lavoro in un sistema operativo. La gestione dei processi comprende aspetti come la loro creazione e cancellazione, nonché la messa a punto di meccanismi per la comunicazione reciproca e la sincronizzazione dei processi. Un sistema operativo, nel gestire la memoria, si cura di registrare quali parti di essa vengano usate e da chi. E sempre al sistema operativo, inoltre, che spetta l'allocazione dinamica e il rilascio dello spazio di memoria. Esso gestisce anche l'archiviazione dei dati: dai file system per i file e le directory, ai dispositivi per la memorizzazione di massa.

Altre due indispensabili funzioni dei sistemi operativi sono le strategie di protezione e le politiche per la sicurezza del sistema. La protezione è garantita da una serie di meccanismi per il controllo dell'accesso, da parte dei processi o degli utenti, alle risorse che il sistema mette a disposizione. Alle misure di sicurezza è invece affidata la difesa dell'elaboratore da attacchi esterni o interni.

I sistemi distribuiti conferiscono agli utenti la possibilità di condividere risorse che giacciono a grande distanza le une dalle altre, su una rete di terminali connessi. I servizi possono essere offerti sia con il modello client-server che con il modello peer-to-peer (ossia, da pari a pari). In un cluster, poiché i dati risiedono in una memoria condivisa, l'elaborazione può essere compiuta da più macchine; di conseguenza, anche se alcuni membri del cluster dovessero subire dei guasti, l'elaborazione può essere portata avanti dagli altri.

I due tipi fondamentali di reti sono le **LAN** e le **WAN**. Le prime consentono l'interazione tra processori collocati in un raggio geografico ristretto, mentre le seconde mettono in comunicazione processori separati da distanze più grandi. Le **LAN** sono generalmente più veloci delle **WAN**.

Vi sono diversi sistemi di calcolo che perseguono scopi specifici. Ne fanno parte i sistemi operativi real-time destinati ad ambienti integrati quali i prodotti commerciali, le automobili e la robotica. I sistemi operativi real-time sono vincolati da scadenze temporali ben definite, che non tollerano ritardi. L'elaborazione deve essere completata entro i limiti prestabiliti, pena il fallimento del sistema. I sistemi multimediali riguardano la trasmissione di dati multimediali, e hanno spesso requisiti particolari per la visualizzazione o la riproduzione di filmati e tracce audio, o casi in cui audio e video sono sincronizzati.

In tempi recenti, l'influenza di Internet e del Web ha incoraggiato lo sviluppo di sistemi operativi moderni che includono come parti integranti elementi quali i browser web, il software per l'accesso alla rete e i programmi per la comunicazione in rete.

SECONDO CAPITOLO:

Strutture dei sistemi operativi

Un SO offre un ambiente in cui eseguire i programmi e fornire servizi.

Si possono individuare alcune classi di servizi offerti dal SO:

—>**interfaccia con l'utente**, può assumere diverse forme(interfaccia a riga di comando che è basata su stringhe che codificano i comandi —> esempio un programma apposito, interfaccia a lotti invece i comandi sono codificati nel file, ed infine l'interfaccia grafica con l'utente cioè un sistema grafico a finestre dotato di un dispositivo puntatore).

—>**esecuzione di un programma**, il sistema deve poter caricare un programma in memoria ed eseguirlo, il programma può terminare in modo normale o anormale segnalando l'errore.

—>**operazioni di input/output**, un programma può richiedere l'uso di un file o di un dispositivo I/O ed è il SO che se ne occupa.

—>**gestione del file System**, I programmi richiedono l'esecuzione di operazioni di lettura e scrittura file, oltre a creare e cancellare file e directory. Alcuni programmi devono anche poter gestire i permessi di accesso ai file sulla base della proprietà del file interessato.

—> **comunicazioni**, alcuni processi hanno bisogno di scambiarsi informazioni ed avviene in due modi: tra processi in esecuzione sullo stesso pc e tra processi in esecuzione su pc diversi collegati tramite una rete. La comunicazione si può realizzare tramite una **memoria condivisa** =(normalmente, il sistema operativo tenta di impedire a un processo l'accesso alla memoria di un altro processo. Il modello a memoria condivisa richiede che più processi concordino nel superare tale limite; a questo punto tali processi possono scambiarsi le informazioni leggendo e scrivendo i dati nelle aree di memoria condivise. La forma e la posizione dei dati sono determinate esclusivamente da questi processi e non sono sotto il controllo del sistema operativo. I processi sono dunque responsabili del rispetto della condizione di non scrivere contemporaneamente nella stessa posizione.—> una variante del modello di processo, detto *thread*, che prevede che la condivisione della memoria sia predefinita.) o attraverso lo **scambio di messaggi** =client/server(trasferisce pacchetti d'informazioni).

Lo scambio di messaggi è utile soprattutto quando è necessario trasferire una piccola quantità di dati, poiché, in questo caso, non sussiste la necessità di evitare conflitti; è inoltre più facile da realizzare rispetto alla condivisione della memoria per la comunicazione tra calcolatori diversi. La condivisione della memoria permette la massima velocità e convenienza nelle comunicazioni, poiché queste ultime, se avvengono all'interno del calcolatore, si svolgono alla velocità della memoria. Sussistono, in ogni caso, problemi per quel che riguarda la protezione e la sincronizzazione tra processi che condividono la memoria.

—>**rilevamento d'errori**, il SO deve essere capace di rilevare eventuali errori che possono verificarsi nella CPU e nei dispositivi di memoria, e deve saper intraprendere l'azione giusta per ciascun tipo d'errore.

Un'altra serie di funzioni del SO che assicura il funzionamento efficiente del sistema stesso:

—>**assegnazione delle risorse**, se sono in corso più sessioni di lavoro di utenti o sono in esecuzione più processi contemporaneamente, il SO provvede all'assegnazione delle risorse necessarie a ciascuno di essi.

—>**contabilizzazione dell'uso delle risorse**, è possibile registrare quali utenti usano il pc segnalando quali e quante risorse impieghino.

—>**protezione e sicurezza**, la protezione assicura che l'accesso alle risorse del sistema sia controllato. La sicurezza di un sistema comincia con l'obbligo d'identificazione da parte di ciascun utente.

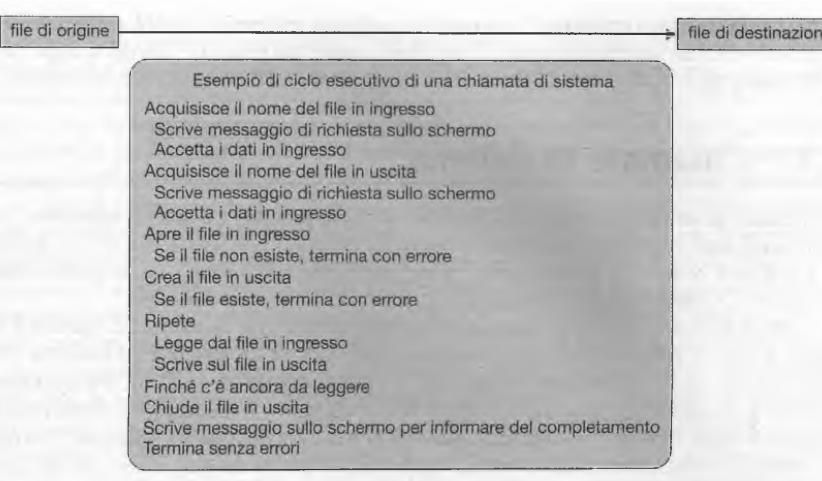
Ci sono due modi per gli utenti di comunicare con il SO:

-interfaccia a riga di comando (interprete dei comandi il quale solitamente è una funzionalità compresa nel kernel)

-interfaccia grafica (strumento più intuitivo/userfriendly) è costituita da una o più finestre.

Chiamate di sistema

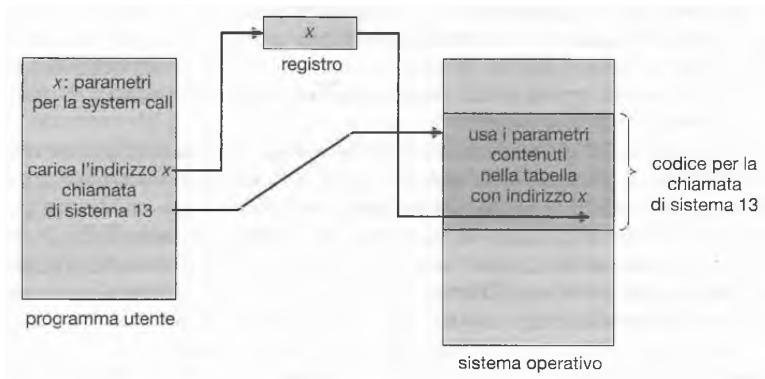
Le **chiamate di sistema** costituiscono l'interfaccia tra un processo e il sistema operativo. Consideriamo come esempio la scrittura di un semplice programma che legga i dati da un file e li trascriva in un altro. Esempio d'uso delle chiamate di sistema.



Come s'è visto, anche programmi molto semplici possono fare un intenso uso del sistema operativo. Non è raro che un sistema esegua migliaia di chiamate di sistema al secondo.

Per passare parametri al sistema operativo si usano tre metodi generali. Il più semplice consiste nel passare i parametri in *registri*; si

possono però presentare casi in cui vi sono più parametri che registri. In questi casi generalmente si memorizzano i parametri in un *blocco* o tabella di memoria e si passa l'indirizzo del blocco, in forma di parametro, in un registro.



• Categorie di chiamate di sistema

Le chiamate di sistema sono classificabili approssimativamente in cinque categorie principali :

- Controllo dei processi
 - terminazione normale e anomala
 - caricamento, esecuzione
 - creazione e arresto di un processo
 - esame e impostazione degli attributi di un processo
 - attesa per il tempo indicato
 - attesa e segnalazione di un evento
 - assegnazione e rilascio di memoria
- Gestione dei file
 - creazione e cancellazione di file
 - apertura, chiusura
 - lettura, scrittura, posizionamento
 - esame e impostazione degli attributi di un file
- Gestione dei dispositivi
 - richiesta e rilascio di un dispositivo
 - lettura, scrittura, posizionamento
 - esame e impostazione degli attributi di un dispositivo
 - inserimento logico ed esclusione logica di un dispositivo
- Gestione delle informazioni
 - esame e impostazione dell'ora e della data
 - esame e impostazione dei dati del sistema
 - esame e impostazione degli attributi dei processi, file e dispositivi
- Comunicazione
 - creazione e chiusura di una connessione
 - invio e ricezione di messaggi
 - informazioni sullo stato di un trasferimento
 - inserimento ed esclusione di dispositivi remoti

Controllo dei processi

Sia in condizioni normali sia anormali il sistema operativo deve trasferire il controllo all'interprete dei comandi che legge il comando successivo. In un sistema interattivo l'interprete dei comandi continua semplicemente a interpretare il comando successivo; si suppone che l'utente invii un comando idoneo per rispondere a qualsiasi errore. In un sistema a interfaccia GUI una finestra avverte l'utente dell'errore e richiede chiarimenti. In un sistema a lotti l'interprete dei comandi generalmente termina il lavoro corrente e prosegue con il successivo.

Programmi di sistema

I programmi di sistema, conosciuti anche come

utilità di sistema, offrono un ambiente più conveniente per lo sviluppo e l'esecuzione dei programmi; alcuni sono semplici interfacce per le chiamate di sistema, altri sono considerevolmente più complessi; in generale si possono classificare nelle seguenti categorie:

—>**Gestione dei file.** Questi programmi creano, cancellano, copiano, ridenominano, stampano, elencano e in genere compiono operazioni sui file e le directory.

—>**Informazioni di stato.** Alcuni programmi richiedono semplicemente al sistema di indicare data, ora, quantità di memoria disponibile o spazio nei dischi, numero degli utenti o informazioni di stato. Altri, più complessi, forniscono informazioni dettagliate su prestazioni, accessi al sistema e debug.

—>**Modifica dei file.** Diversi editor sono disponibili per creare e modificare il contenuto di file memorizzati su dischi o altri dispositivi, oltre a comandi speciali per l'individuazione di contenuti di file o per particolari trasformazioni del testo.

—>**Ambienti d'ausilio alla programmazione.** Compilatori, assemblatori, programmi per la correzione degli errori e interpreti dei comuni linguaggi di programmazione

—>**Caricamento ed esecuzione dei programmi.** Una volta assemblato o compilato, per essere eseguito, un programma deve essere caricato in memoria. Il sistema può mettere a disposizione caricatori assoluti, caricatori rilocabili, editor dei collegamenti (*linkage editor*) e caricatori di sezioni sovrapponibili di programmi (*overlay loader*).

—>**Comunicazioni.** Questi programmi offrono i meccanismi con cui si possono creare collegamenti virtuali tra processi, utenti e calcolatori diversi.

PROGETTARE UN SO:

Costruzione del SO con il microkernel—>offre maggiori garanzie di sicurezza e affidabilità, poiché i servizi si eseguono in gran parte come processi utenti, e non come processi del kernel: se un servizio è compromesso, il resto del sistema operativo rimane intatto. Purtroppo i microkernel possono incorrere in cali di prestazioni dovuti al sovraccarico indotto dall'esecuzione di processi utente con funzionalità di sistema. Forse il miglior approccio attualmente disponibile per la progettazione dei sistemi operativi si fonda su tecniche della programmazione orientata agli oggetti per implementare un kernel modulare. Rispetto ai sistemi orientati a un microkernel,

l'efficienza è superiore, perché i moduli sono in grado di comunicare senza invocare le funzionalità di trasmissione dei messaggi.

Debugging dei sistemi operativi

Il **debugging** può essere genericamente definito come l'attività di individuare e risolvere errori nel sistema, i cosiddetti **bachi** (*bugs*). Questa operazione viene effettuata sia sull'hardware sia sul software.

Sommario 2° CAPITOLO

I sistemi operativi offrono diversi servizi: al livello più basso, le chiamate di sistema permettono al programma in esecuzione di fare richieste direttamente al sistema operativo; a un livello superiore, l'interprete dei comandi (in alcuni ambiti noto come *shelt*) mette a disposizione un meccanismo che consente a un utente di impartire una richiesta senza scrivere un programma. I comandi possono provenire da file, in un'esecuzione a lotti (*batch*) oppure direttamente da una tastiera, in modo interattivo o a partizione del tempo. I programmi di sistema offrono agli utenti i servizi più comuni.

I tipi di richieste variano secondo il livello delle stesse richieste. Il livello cui appartengono le chiamate di sistema deve offrire le funzioni di base, come quelle di controllo dei processi e gestione di file e dispositivi. Le richieste di livello superiore, soddisfatte dall'interprete dei comandi o dai programmi di sistema, sono tradotte in una sequenza di chiamate di sistema. I servizi di sistema si possono classificare in diverse categorie: controllo dei programmi, richieste di stato e richieste di I/O. Gli errori dei programmi si possono considerare richieste di servizio implicite. Una volta definiti i servizi del sistema è possibile passare allo sviluppo della struttura del sistema operativo. Per registrare le informazioni che definiscono lo stato del calcolatore e lo stato dei processi del sistema occorrono diverse tabelle. La progettazione di un nuovo sistema operativo è un compito molto difficile. Gli scopi del sistema si devono definire chiaramente prima di iniziare la progettazione; costituiscono la base da cui partire per poter scegliere tra le varie strategie e i vari algoritmi necessari. Poiché un sistema operativo è di grandi dimensioni, la modularità è un altro fattore importante. La progettazione di un sistema come una sequenza di strati o l'uso di un microkernel sono considerati buone tecniche. Il concetto di macchina virtuale tiene in grande considerazione il metodo basato sulla stratificazione e tratta il kernel del sistema operativo come se facesse parte della macchina fisica. Su questa macchina virtuale si possono caricare persino altri sistemi operativi. In tutto il ciclo di progettazione del sistema operativo, occorre prestare attenzione alla distinzione tra la scelta dei criteri e i dettagli dei meccanismi adottati. In questo modo si ottiene la massima flessibilità, che all'occorrenza consente di modificare più facilmente i criteri adottati. Ormai i sistemi operativi sono quasi tutti scritti in un linguaggio per lo sviluppo di sistemi o in un linguaggio di alto livello; questa caratteristica facilita realizzazione, manutenzione e adattabilità a sistemi diversi. Il processo di debugging e i guasti nel kernel possono essere studiati grazie all'utilizzo di debugger e di altri strumenti in grado di analizzare un'immagine dello stato della memoria. Strumenti quali DTrace analizzano i sistemi di produzione per trovare colli di bottiglia e capire altri comportamenti del sistema. All'avvio di un calcolatore, la CPU deve eseguire il programma d'avvio residente nel firmware. Se l'intero sistema operativo risiede nel firmware, all'accensione l'intero sistema è eseguibile direttamente; altrimenti, il ciclo di avvio della macchina procede per fasi progressive, a ognuna delle quali si caricano in memoria, dal firmware e dal disco, porzioni sempre più potenti del sistema operativo, fino a eseguire l'intero sistema stesso.

TERZO CAPITOLO

Gestione dei processi

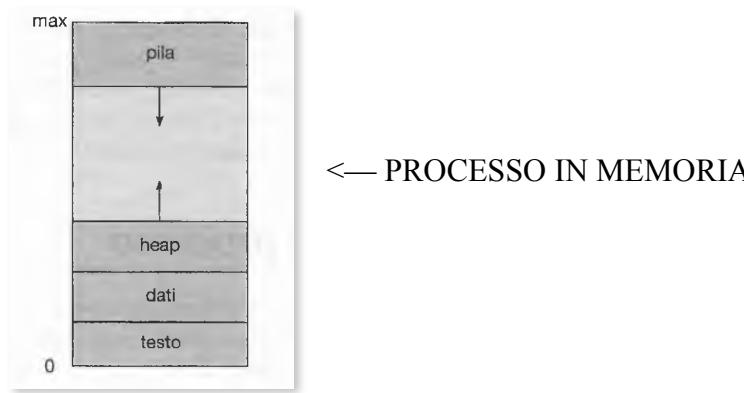
Un *processo* si può pensare come un programma in esecuzione. Per svolgere il proprio compito, un processo richiede determinate risorse, come tempo d'elaborazione della CPU, memoria, file e dispositivi di I/O. Queste risorse si assegnano al processo al momento della sua creazione o durante l'esecuzione. I primi sistemi di calcolo consentivano l'esecuzione di un solo programma alla volta, che aveva il completo controllo del sistema e accesso a tutte le sue risorse. Gli attuali sistemi di calcolo consentono, invece, che più programmi siano caricati in memoria ed eseguiti in modo concorrente. Tale evoluzione richiede un più severo controllo e una maggiore compartimentazione dei vari programmi. Da tali necessità deriva la nozione di **processo d'elaborazione** —o, più

brevemente, **processo** - che in prima istanza si può definire come un programma in esecuzione, e costituisce l'unità di lavoro dei moderni sistemi a partizione del tempo d'elaborazione. Un sistema è quindi costituito da un insieme di processi: quelli del sistema operativo eseguono il codice di sistema; gli utenti il codice utente. Tutti questi processi si possono eseguire potenzialmente in modo concorrente e l'uso della CPU (o di più unità d'elaborazione) è commutato tra i vari processi.

CONCETTO DI PROCESSO

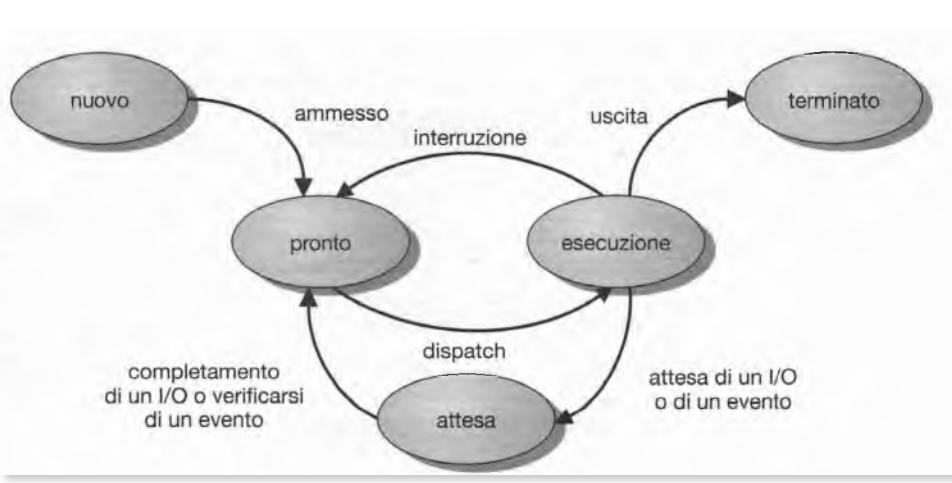
Una questione che sorge dall'analisi dei sistemi operativi è la definizione delle attività della CPU. Un **sistema a lotti (batch)** esegue **lavori (job)**, mentre un sistema a partizione del tempo esegue **programmi utenti o task**.

Un *processo* è un programma in esecuzione. È qualcosa di più del codice di un programma, talvolta noto anche come **sezione di testo**: comprende l'attività corrente, rappresentata dal valore del **contatore di programma** e dal contenuto dei registri della CPU; normalmente comprende anche la propria **pila (.stack)**, contenente a sua volta i dati temporanei, come i parametri di un metodo, gli indirizzi di rientro e le variabili locali, e una **sezione di dati** contenente le variabili globali. Un processo può includere uno **heap**, ossia della memoria dinamicamente allocata durante l'esecuzione del processo.



Stato del processo

Un processo durante l'esecuzione è soggetto a cambiamenti di stato, definiti in parte dalla attività corrente del processo stesso. Ogni processo può trovarsi in uno tra i seguenti stati.



- **Nuovo**-Si crea il processo.
- **Esecuzione**-Un'unità d'elaborazione esegue le istruzioni del relativo programma.
- **Attesa**. Il processo attende che si verifichi qualche evento (come il completamento di un'operazione di i/o o la ricezione di un segnale).
- **Pronto**. Il processo attende di essere assegnato a un'unità d'elaborazione.
- **Terminato**. Il processo ha terminato l'esecuzione.

Blocco di controllo dei processi

Ogni processo è rappresentato nel sistema operativo da un blocco di controllo di un processo (*process control block*, PCB, o *task control block*, TCB). Un blocco di controllo di un processo (Figura 3.3) contiene molte informazioni connesse a un processo specifico, tra cui le seguenti.

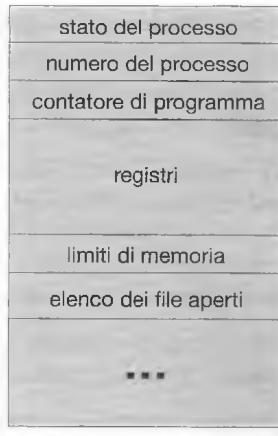


Figura 3.3 Blocco di controllo di un processo (PCB).

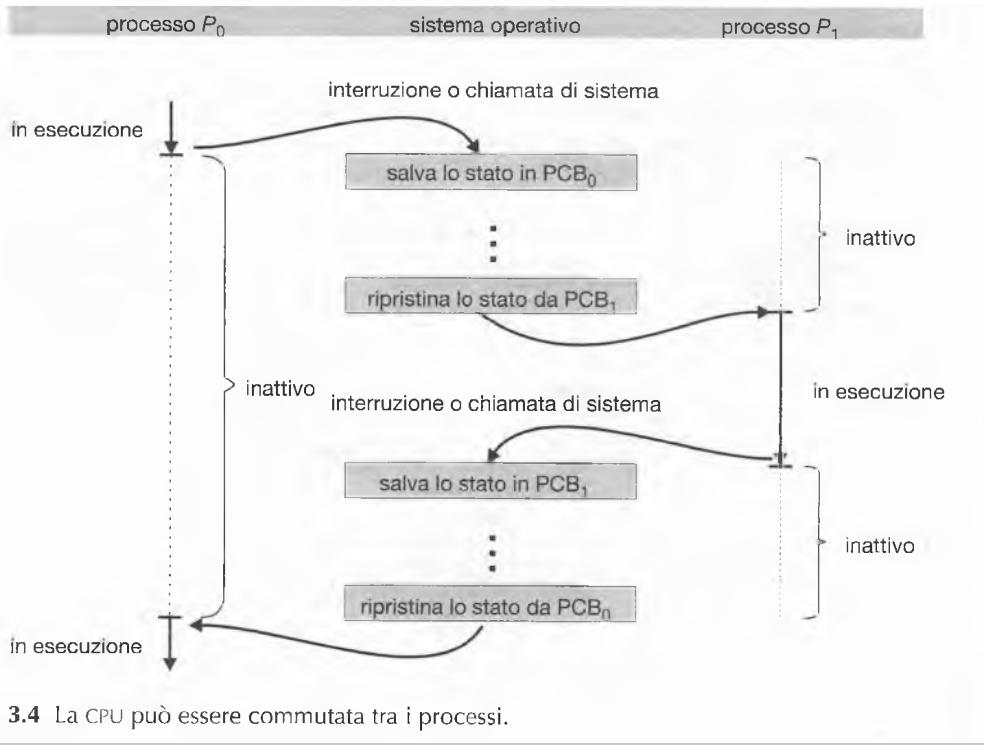


Figura 3.4 La CPU può essere commutata tra i processi.

—>**Stato del processo.** Lo stato può essere: nuovo, pronto, esecuzione, attesa, arresto, e così via.

—>**Contatore di programma.** Il contatore di programma contiene l'indirizzo della successiva istruzione da eseguire per tale processo.

—>**Registri di CPU.** I registri variano in numero e tipo secondo l'architettura del calcolatore. Essi

comprendono accumulatori, registri d'indice, puntatori alla cima delle strutture a pila (*stackpointer*), registri d'uso generale e registri contenenti informazioni relative ai codici di condizione. Quando si verifica un'interruzione della CPU, si devono salvare tutte queste informazioni insieme con il contatore di programma, in modo da permettere la corretta esecuzione del processo in un momento successivo .

—>**Informazioni sullo scheduling di CPU.** Queste informazioni comprendono la priorità del processo, i puntatori alle code di scheduling e tutti gli altri parametri di scheduling.

—>**Informazioni sulla gestione della memoria.** Queste informazioni si possono esprimere attraverso i valori dei registri di base e di limite, le tabelle delle pagine o le tabelle dei segmenti, a seconda del sistema di gestione della memoria usato dal sistema operativo (Capitolo 8).

—>**Informazioni di contabilizzazione delle risorse.** Queste informazioni comprendono il tempo d'uso della CPU e il tempo reale d'utilizzo della stessa, i limiti di tempo, i numeri dei processi, e così via.

—>**Informazioni sullo stato dell'I/O.** Queste informazioni comprendono la lista dei dispositivi di I/O assegnati a un determinato processo, l'elenco dei file aperti, e così via.

In sintesi, il PCB si usa semplicemente come deposito per tutte le informazioni relative ai vari processi.

Thread:

Il modello dei processi illustrato fin qui sottintende che un processo è un programma che si esegue seguendo un unico percorso d'esecuzione, detto **thread**. Se un processo sta, per esempio, eseguendo un programma di elaborazione di testi, l'esecuzione avviene seguendo una singola sequenza di istruzioni; quindi il processo può svolgere un solo compito alla volta. Tramite lo stesso processo, per esempio, un utente non può contemporaneamente inserire caratteri e verificare la correttezza ortografica di quel che sta scrivendo. In molti sistemi operativi moderni si è esteso il concetto di processo introducendo la possibilità d'avere più corsi d'esecuzione, in modo da permettere che un processo possa svolgere più di un compito alla volta.

Scheduling dei processi

L'obiettivo della multiprogrammazione consiste nel disporre dell'esecuzione contemporanea di alcuni processi in modo da massimizzare l'utilizzo della CPU. L'obiettivo della partizione del tempo è di commutare l'uso della CPU tra i vari processi così frequentemente che gli utenti possano interagire con ciascun programma mentre è in esecuzione. Per raggiungere questi obiettivi, lo **scheduler dei processi** seleziona un processo da eseguire dall'insieme di quelli disponibili. Nei sistemi monoprocesso non vi sarà mai più di un processo in esecuzione: gli altri dovranno attendere finché la CPU sia nuovamente disponibile.

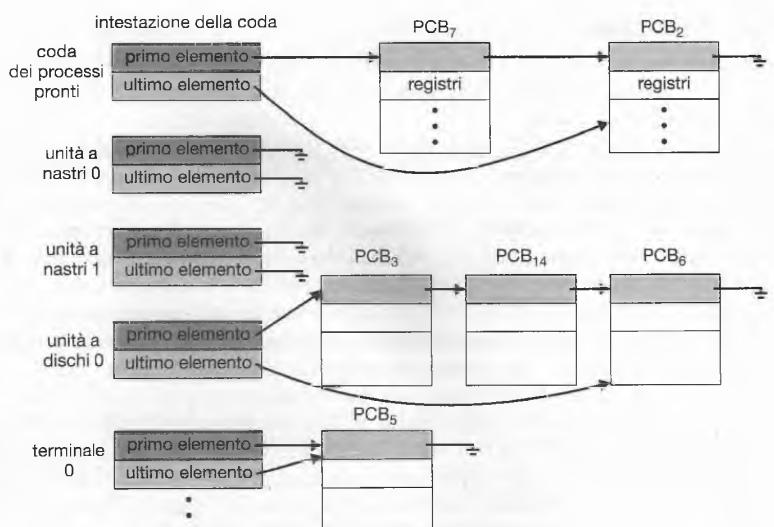
Code di scheduling

Ogni processo è inserito in una **coda di processi**, composta da tutti i processi del sistema. I processi presenti in memoria centrale, che sono pronti e nell'attesa d'essere eseguiti, si trovano in una lista detta **coda dei processi pronti** (*ready queue*). Questa coda generalmente si memorizza come una lista concatenata: un'intestazione della coda dei processi pronti con tiene i puntatori al primo e all'ultimo PCB dell'elenco, e ciascun PCB è esteso con un

campo puntatore che indica il successivo processo contenuto nella coda dei processi pronti.

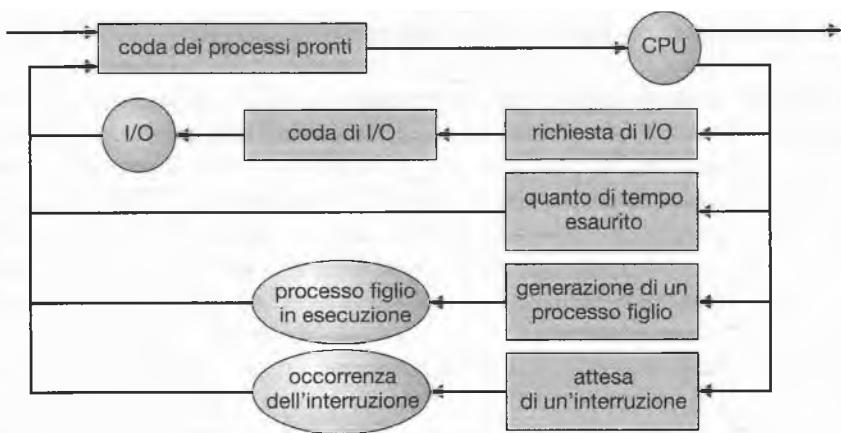
Il sistema operativo ha anche altre code. Quando si assegna la CPU a un processo, quest'ultimo rimane in esecuzione per un certo tempo e prima o poi termina, viene interrotto, oppure si ferma nell'attesa di un evento particolare, come il completamento di una richiesta

di I/O. Una richiesta di i/o può essere diretta a un dispositivo condiviso, come un disco. Poiché nel sistema esistono molti processi, il disco può essere occupato con una richiesta di I/O di qualche altro processo, quindi il processo deve attendere che il disco sia disponibile.



3.6 Coda dei processi pronti e diverse code di dispositivi I/O.

Una comune rappresentazione utile alla descrizione dello scheduling dei processi è data da un **diagramma di accodamento** come quello illustrato nella Figura 3.7.



3.7 Diagramma di accodamento per lo scheduling dei processi.

Ogni riquadro rappresenta una coda. Sono presenti due tipi di coda: la coda dei processi pronti e un insieme di code di dispositivi. I cerchi rappresentano le risorse che servono le code, le frecce indicano il flusso di processi nel sistema. Un nuovo processo si colloca inizialmente nella coda dei processi pronti, dove attende finché non è selezionato per essere eseguito (*dispatched*). Una volta che il processo è assegnato alla CPU ed è nella fase d'esecuzione, si può verificare uno dei seguenti eventi:

- ♦ il processo può emettere una richiesta di I/O

e quindi essere inserito in una coda di I/O;

- ♦ il processo può creare un nuovo processo e attenderne la terminazione;
- ♦ il processo può essere rimosso forzatamente dalla CPU a causa di un'interruzione, ed essere reinserito nella coda dei processi pronti.

Nei primi due casi, al completamento della richiesta di I/O o al termine del processo figlio, il processo passa dallo

stato d'attesa allo stato pronto ed è nuovamente inserito nella coda dei processi pronti. Un processo continua questo ciclo fino al termine della sua esecuzione: a questo punto viene allontanato da tutte le code, rimosso il suo PCB e revocate le varie risorse.

Nel corso della sua esistenza, un processo si trova in varie code di scheduling. Il sistema operativo, incaricato di selezionare i processi dalle suddette code, compie la selezione per mezzo di un opportuno **scheduler**.

Spesso, in un sistema a lotti, accade che si sottopongano più processi di quanti se ne possano eseguire immediatamente. Questi lavori si trasferiscono in dispositivi di memoria secondaria, generalmente dischi, dove si tengono fino al momento dell'esecuzione (*spooling*). Lo **scheduler a lungo termine** (*job scheduler*), sceglie i lavori da questo insieme e li carica in memoria affinché siano eseguiti. Lo **scheduler a breve termine**, o **scheduler di CPU**, fa la selezione tra i lavori pronti per l'esecuzione e assegna la CPU a uno di loro. Questi due scheduler si differenziano principalmente per la frequenza con la quale sono eseguiti. Lo scheduler a breve termine seleziona frequentemente un nuovo processo per la CPU. Lo scheduler a lungo termine, invece, si esegue con una frequenza molto inferiore; di versi minuti possono trascorrere tra la creazione di un nuovo processo e il successivo. Lo scheduler a lungo termine controlla il **grado di multiprogrammazione**, cioè il numero di processi presenti in memoria. Se è stabile, la velocità media di creazione dei processi deve essere uguale alla velocità media con cui i processi abbandonano il sistema; quindi lo scheduler a lungo termine si può richiamare solo quando un processo abbandona il sistema. A causa del maggior intervallo che intercorre tra le esecuzioni, lo scheduler a lungo termine dispone di più tempo per scegliere un processo per l'esecuzione. Le prestazioni migliori sono date da una combinazione equilibrata di processi con prevalenza di I/O e processi con prevalenza d'elaborazione.

Cambio di contesto/CONTEXT SWITCH

Sono le interruzioni a indurre il sistema a sospendere il lavoro attuale della CPU per eseguire routine del kernel. Le interruzioni sono eventi comuni nei sistemi a carattere generale. In presenza di una interruzione, il sistema deve salvare il **contesto** del processo corrente, per poterlo poi ripristinare quando il processo stesso potrà ritornare in esecuzione.

Il contesto è rappresentato all'interno del PCB del processo, e com prende i valori dei registri della CPU, lo stato del processo e informazioni relative alla gestione della memoria. In termini generali, si esegue un **salvataggio dello stato** corrente della CPU, sia che essa esegua in modalità utente o in modalità di sistema; in seguito, si attuerà un corrispondente **ripristino dello stato** per poter riprendere l'elaborazione dal punto in cui era stata interrotta.

Il passaggio della CPU a un nuovo processo implica la registrazione dello stato del precedente vecchio e il caricamento dello stato precedentemente registrato del nuovo processo. Questa procedura è nota col nome di **cambio di contesto** (*context switch*).

Nell'evenienza di un cambio di contesto, il sistema salva nel suo PCB il contesto del processo uscente, e carica il contesto del processo subentrante, salvato in precedenza. Il cambio di contesto comporta un calo delle prestazioni, perché il sistema esegue solo operazioni volte alla corretta gestione dei processi, e non alla computazione.

Operazioni sui processi

Nella maggior parte dei sistemi i processi si possono eseguire in modo concorrente, e si devono creare e cancellare dinamicamente; a tal fine il sistema operativo deve offrire un meccanismo che permetta di creare e terminare un processo.

CREAZIONE DEL PROCESSO

Durante la propria esecuzione, un processo può creare numerosi nuovi processi tramite un'apposita chiamata di sistema (*create_process*). Il processo creante si chiama processo **genitore**, mentre il nuovo processo si chiama processo **figlio**. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando un **albero** di processi.

Quando un processo ne crea uno nuovo, per quel che riguarda l'esecuzione ci sono due possibilità:

1. il processo genitore continua l'esecuzione in modo concorrente con i propri processi figli;
2. il processo genitore attende che alcuni o tutti i suoi processi figli terminino.

Ci sono due possibilità anche per quel che riguarda lo spazio d'indirizzi del nuovo processo:

1. il processo figlio è un duplicato del processo genitore;
2. nel processo figlio si carica un nuovo programma.

TERMINAZIONE DI UN PROCESSO

Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la chiamata di sistema `exit()`; a questo punto, il processo figlio può riportare alcuni dati al processo genitore, che li riceve attraverso la chiamata di sistema `wait()`. Tutte le risorse del processo, incluse la memoria fisica e virtuale, i file aperti e le aree della memoria per l'i/O, sono liberate dal sistema operativo. Generalmente solo il genitore del processo che si vuole terminare può invocare una chiamata di sistema di questo tipo, altrimenti gli utenti potrebbero causare arbitrariamente la terminazione forzata di processi di chiunque. Occorre notare che un genitore deve conoscere le identità dei propri figli, perciò quando un processo ne crea uno nuovo, l'identità del nuovo processo viene passata al processo genitore. Un processo genitore può porre termine all'esecuzione di uno dei suoi processi figli per diversi motivi, tra i quali i seguenti.

—> Il processo figlio ha ecceduto nell'uso di alcune tra le risorse che gli sono state assegnate. Ciò richiede che il processo genitore disponga di un sistema che esamini lo stato dei propri processi figli.

—> Il compito assegnato al processo figlio non è più richiesto.

—> Il processo genitore termina e il sistema operativo non consente a un processo figlio di continuare l'esecuzione in tale circostanza.

COMUNICAZIONE TRA PROCESSI

I processi eseguiti concorrentemente nel sistema operativo possono essere indipendenti o cooperanti. Un processo è **indipendente** se non può influire su altri processi del sistema o subirne l'influsso. Chiaramente, un processo che non condivide dati (temporanei o permanenti) con altri processi è indipendente. D'altra parte un processo è **cooperante** se influenza o può essere influenzato da altri processi in esecuzione nel sistema. Ovviamente, qualsiasi processo che condivide dati con altri processi è un processo cooperante.

Sommario 3°CAPITOLO

Un processo è un programma in esecuzione. Nel corso delle sue attività, un processo cambia stato, e tale stato è definito dall'attività corrente del processo stesso. Ogni processo può trovarsi in uno tra i seguenti stati: nuovo, pronto, esecuzione, attesa o arresto. In un sistema operativo ogni processo è rappresentato dal proprio blocco di controllo del processo (PCB).

Un processo, quando non è in esecuzione, è inserito in una coda d'attesa. Le due classi principali di code in un sistema operativo sono le code di richieste di I/O e la coda dei processi pronti per l'esecuzione, quest'ultima contenente tutti i processi pronti per l'esecuzione che si trovano in attesa della CPU. Ogni processo è rappresentato da un PCB; i PCB si possono collegare tra loro in modo da formare una coda dei processi pronti. Lo scheduling a lungo termine (*o job scheduling*) consiste nella scelta dei processi che si contenderanno la CPU. Normalmente lo scheduling a lungo termine è influenzato in modo consistente da considerazioni riguardanti l'assegnazione delle risorse, in particolar modo quelle concernenti la gestione della memoria. Lo scheduling a breve termine (*o scheduling della CPU*) consiste nella selezione di un processo dalla coda dei processi pronti.

I sistemi operativi devono implementare un meccanismo per generare processi figli da un processo genitore. Genitore e figli possono girare in concomitanza, oppure il genitore potrà essere posto in attesa della terminazione dei figli. L'esecuzione concorrente è ampiamente giustificabile dalla necessità di condividere informazioni e dal potenziale aumento della velocità di calcolo, oltre che da considerazioni legate alla modularità e alla convenienza.

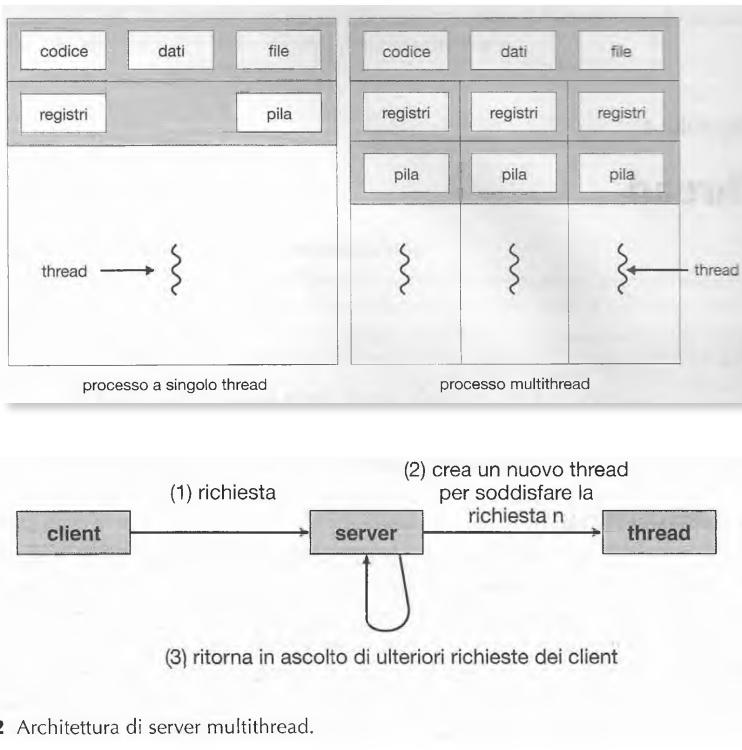
I processi in esecuzione nel sistema operativo possono essere indipendenti o cooperanti. I processi cooperanti devono avere i mezzi per comunicare tra loro. Fondamentalmente esistono due schemi complementari di comunicazione: memoria condivisa e scambio di messaggi. Nel metodo con memoria condivisa i processi in comunicazione devono condividere alcune variabili, per mezzo di cui i processi scambiano informazioni; il compito della comunicazione è lasciato ai programmati di applicazioni; il sistema operativo deve semplicemente offrire la memoria condivisa. Il metodo con scambio di messaggi permette di compiere uno scambio di messaggi tra i processi; in questo caso il compito di attuare la comunicazione è del sistema operativo. Questi due schemi non sono mutuamente esclusivi, e si possono impiegare insieme in uno stesso sistema.

La comunicazione nei sistemi client/server può impiegare (1) socket, (2) chiamate di procedure remote (RPC) o (3) chiamate di metodi remoti (RMI). Una connessione tra una coppia di applicazioni consiste di una coppia di socket, ciascuna a un'estremità del canale di comunicazione. Le RPC sono un'altra forma di comunicazione distribuita: una RPC si verifica quando un processo (o un thread) invoca una procedura in un'applicazione remota. Le RMI sono la

versione del linguaggio Java delle RPC; consentono a un thread di invocare un metodo su un oggetto remoto esattamente come se si trattasse di oggetto locale. La differenza principale tra le RPC e le RMI consiste nel fatto che i dati passati a una procedura remota hanno la forma di un'ordinaria struttura dati, mentre le RMI consentono anche il passaggio di oggetti.

QUARTO CAPITOLO

Thread



4.2 Architettura di server multithread.

Nel modello introdotto nel Capitolo 3 un processo è considerato come un programma in esecuzione con un unico percorso di controllo. Molti sistemi operativi moderni permettono che un processo possa avere più percorsi di controllo che comunemente si chiamano *thread*. Un thread è l'unità di base d'uso della CPU e comprende un identificatore di thread (ID), un contatore di programma, un insieme di registri, e una pila (*.stack*). Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali. Un processo tradizionale, chiamato anche **processo pesante** (*heavyweightprocess*) , è composto da un solo thread. Un processo multithread è in grado di lavorare a più compiti in modo concorrente.

Vantaggi

I **vantaggi** della programmazione multithread si possono classificare rispetto a quattro fattori principali.

1. **Tempo di risposta.** Rendere multithread un'applicazione interattiva può permettere a un programma di continuare la sua esecuzione, anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo di risposta medio all'utente. Per esempio, un programma di consultazione del Web multithread potrebbe permettere l'interazione con l'utente tramite un thread mentre un'immagine verrebbe caricata da un altro thread.
2. **Condivisione delle risorse.** I processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa o il passaggio di messaggi. Queste tecniche devono essere esplicitamente messe in atto e organizzate dal programmatore. Tuttavia, i thread condividono d'ufficio la memoria e le risorse del processo al quale appartengono. Il vantaggio della condivisione del codice consiste nel fatto che un'applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d'indirizzi.
3. **Economia.** Assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambi di contesto. È difficile misurare empiricamente la differenza del carico richiesto per creare e gestire un processo invece che un thread, tuttavia la creazione e gestione dei processi richiede in generale molto più tempo. In Solaris, per esempio, la creazione di un processo richiede un tempo trenta volte maggiore di quello richiesto per la creazione di un thread, un cambio di contesto per un processo richiede un tempo pari a circa cinque volte quello richiesto per un thread.
4. **Scalabilità.** I vantaggi della programmazione multithread aumentano notevolmente nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo (uno per ciascun processore). Un processo con un singolo thread può funzionare solo su un processore, indipendentemente da quanti ve ne siano a disposizione. Il multithreading su una macchina con più processori incrementa il parallelismo. Esploreremo questo tema nel prossimo paragrafo.

Programmazione multicore

Una tendenza recente nel progetto dell'architettura dei sistemi consiste nel montare diverse unità di calcolo (*core*) su un unico processore (un processore *multicore*); ogni unità appare al sistema operativo come un processore separato. La programmazione multi thread offre un meccanismo per un utilizzo più efficiente di questi processori e aiuta a sfruttare al meglio la concorrenza. Si consideri un'applicazione con quattro thread.

In un sistema con una singola unità di calcolo "esecuzione concorrente" significa solo che l'esecuzione dei thread è stratificata nel tempo, o come anche si dice, *interfogliata (interleaved)* (Figura 4.3), perché la CPU è in grado di eseguire un solo thread alla volta. Su un sistema multicore, invece, "esecuzione concorrente" significa che i thread possono funzionare in parallelo, dal momento che il sistema può assegnare thread diversi a ciascuna unità di calcolo (Figura 4.4).

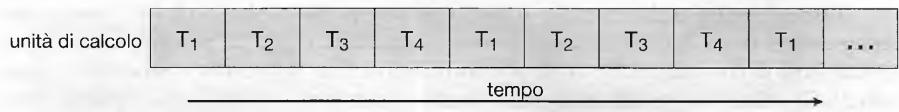


Figura 4.3 Esecuzione concorrente su un sistema a singola unità di calcolo.

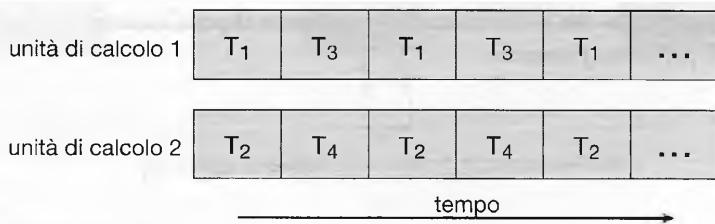


Figura 4.4 Esecuzione parallela su un sistema multicore.

In generale, possiamo individuare nelle cinque aree seguenti i principali obiettivi della programmazione dei sistemi multicore.

1. **Separazione dei task.** Consiste nell'esaminare le applicazioni al fine di individuare aree separabili in task distinti e concorrenti che possano essere eseguiti in parallelo su unità di calcolo distinte.
2. **Bilanciamento.** Nell'identificare i task eseguibili in parallelo, i programmati devono far sì che i vari task eseguano compiti di mole e valore confrontabili. In alcuni casi si verifica che un determinato task non contribuisca al processo complessivo tanto quanto gli altri; in questi casi per eseguire il task può non valere la pena utilizzare un'unità di calcolo separata.
3. **Suddivisione dei dati.** Proprio come le applicazioni sono divise in task separati, i dati a cui i task accedono, e che manipolano, devono essere suddivisi per essere utilizzati da unità di calcolo distinte.
4. **Dipendenze dei dati.** I dati a cui i task accedono devono essere esaminati per verificare le dipendenze tra due o più task. In esempi in cui un task dipende dai dati forniti da un altro, i programmati devono assicurare che l'esecuzione dei task sia sincronizzata in modo da soddisfare queste dipendenze.
5. **Test e debugging.** Quando un programma funziona in parallelo su unità multiple, vi sono diversi possibili flussi di esecuzione. Effettuare i test e il debugging di programmi concorrenti è per natura più difficile rispetto al caso di applicazioni con un singolo thread.

I thread possono essere distinti in thread a livello utente e thread a livello kernel: i primi sono gestiti senza l'aiuto del kernel; i secondi, invece, sono gestiti direttamente dal sistema operativo. Un segnale si può ricevere in modo sincrono o asincrono, secondo la sorgente e la ragione della segnalazione dell'evento. Indipendentemente dal modo di ricezione sincrono o asincrono, tutti i segnali seguono lo stesso schema:

1. all'occorrenza di un particolare evento si genera un segnale;
2. s'invia il segnale a un processo;
3. una volta ricevuto, il segnale deve essere gestito.

Un accesso illegale alla memoria o una divisione per zero generano segnali sincroni. In questi casi, se un programma in esecuzione compie le suddette azioni, viene generato un segnale. I segnali sincroni s'inviano allo stesso processo che ha eseguito l'operazione causa del segnale (questo è il motivo per cui si chiamano sincroni). Quando un segnale è causato da un evento esterno al processo in esecuzione, tale processo riceve il segnale in modo asincrono.

Sommario 4°CAPITOLO

Un thread è un percorso di controllo d'esecuzione all'interno di un processo. Un processo multithread contiene più percorsi di controllo diversi, ma che condividono lo stesso spazio d'indirizzi. I vantaggi della programmazione multithread includono un miglioramento del tempo di risposta, la condivisione di risorse all'interno del processo, il risparmio e la capacità di sfruttare le architetture dotate di più unità d'elaborazione. I thread a livello utente sono thread visibili al programmatore e sconosciuti al kernel. Il kernel del sistema operativo gestisce thread a livello kernel. In generale, i thread a livello utente richiedono minor tempo per essere creati e gestiti rispetto a quelli a livello kernel, senza necessità d'intervento da parte del kernel. Ci sono tre tipi diversi di modelli che descrivono le relazioni fra thread a livello utente e a livello kernel: il modello da molti a uno associa più thread a livello utente a un singolo thread a livello kernel; il modello da uno a uno associa ciascun thread a livello utente a un corrispondente thread a livello kernel; il modello da molti a molti associa dinamicamente più thread a livello utente a un numero minore o uguale di thread a livello kernel.

Molti sistemi operativi moderni prevedono la gestione dei thread a livello kernel: tra questi i sistemi Windows 98, NT, 2000 e XP, oltre a Solaris e Linux. Per la creazione e la gestione dei thread le relative librerie forniscono una API al programmatore di applicazioni. Le tre più comuni librerie di thread sono: POSIX Pthread, Win32 per i sistemi Windows e i thread Java. I programmi multithread presentano molti aspetti critici per il programmatore, tra cui la semantica delle chiamate di sistema fork () ed exec (); altri aspetti sono per esempio la cancellazione, la gestione dei segnali e i dati privati dei thread.

CAPITOLO CINQUE

Scheduling della CPU

Lo scheduling della CPU è alla base dei sistemi operativi multiprogrammati: attraverso la commutazione del controllo della CPU tra i vari processi, il sistema operativo può rendere più produttivo il calcolatore. In questo capitolo s'introducono i concetti fondamentali dello scheduling e si descrivono vari algoritmi di scheduling della CPU. Si affronta inoltre il problema della scelta dell'algoritmo da impiegare per un dato sistema.

In un sistema monoprocesso si può eseguire al massimo un processo alla volta; gli altri processi, se ci sono, devono attendere che la CPU sia libera e possa essere nuovamente sotto posta a scheduling.

L'idea della multiprogrammazione è relativamente semplice. Un processo è in esecuzione finché non deve attendere un evento, generalmente il completamento di qualche richiesta di I/O; durante l'attesa, in un sistema di calcolo semplice, la CPU resterebbe inattiva, e tutto il tempo d'attesa sarebbe sprecato. Con la multiprogrammazione si cerca d'impiegare questi tempi d'attesa in modo produttivo: si tengono contemporaneamente più processi in memoria, e quando un processo deve attendere un evento, il sistema operativo gli sottrae il controllo della CPU per cederlo a un altro processo. Lo scheduling è una funzione fondamentale dei sistemi operativi; si sottopongono a scheduling quasi tutte le risorse di un calcolatore. Naturalmente la CPU è una delle risorse principali, e il suo scheduling è alla base della progettazione dei sistemi operativi.

Ciclicità delle fasi d'elaborazione e di I/O:

Il successo dello scheduling della CPU dipende dall'osservazione della seguente proprietà dei processi: l'esecuzione del processo consiste in un ciclo d'elaborazione (svolta dalla CPU) e d'attesa del completamento delle operazioni di I/O. I processi si alternano tra questi due stati. L'esecuzione di un processo comincia con una sequenza (una "raffica") di operazioni d'elaborazione svolte dalla CPU (*CPU burst*), seguita da una sequenza di operazioni di I/O (*i/O burst*), quindi un'altra sequenza di operazioni della CPU, di nuovo una sequenza di operazioni di I/O, e così via. L'ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l'esecuzione (Figura 5.1). Le durate delle sequenze di operazioni della CPU sono state misurate, e sebbene varino molto secondo il processo e secondo il calcolatore, la loro curva di frequenza è simile a quella illustrata nella Figura 5.2. La curva è generalmente di tipo esponenziale o iperbolico, con molte brevi sequenze di operazioni della CPU, e poche sequenze di operazioni della CPU molto lunghe. Un programma con prevalenza di I/O (*I/O bound*) produce generalmente molte sequenze di operazioni della CPU di breve durata. Un programma con prevalenza d'elaborazione (*CPU bound*), invece, produce poche sequenze di operazioni della CPU molto lunghe. Queste



5.2 Diagramma delle durate delle sequenze di operazioni della CPU.

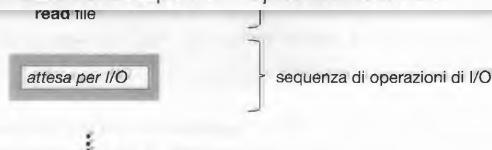


Figura 5.1 Serie alternata di sequenze di operazioni della CPU e di sequenze di operazioni di I/O.

nella lista d'attesa per accedere alla CPU. Generalmente gli elementi delle code sono i *process control block* (PCB) dei processi.

Scheduling con diritto di prelazione :

Le decisioni riguardanti lo scheduling della CPU si possono prendere nelle seguenti circostanze:

1. un processo passa dallo stato di esecuzione allo stato di attesa (per esempio, richiesta di I/O o richiesta di attesa (wait) per la terminazione di uno dei processi figli);
2. un processo passa dallo stato di esecuzione allo stato pronto (per esempio, quando si verifica un segnale d'interruzione);
3. un processo passa dallo stato di attesa allo stato pronto (per esempio, al completamento di un'operazione di I/O);
4. un processo termina.

I casi 1 e 4 in quanto tali non comportano alcuna scelta di scheduling; a essi segue la scelta di un nuovo processo da eseguire, sempre che ce ne sia almeno uno nella coda dei processi pronti per l'esecuzione. Una scelta si deve invece fare nei casi 2 e 3.

Quando lo scheduling interviene solo nelle condizioni 1 e 4, si dice che lo schema di scheduling è senza diritto di prelazione (*nonpreemptive*) o cooperativo (*cooperative*); altrimenti, lo schema di scheduling è con diritto di prelazione (*preemptive*). Nel caso dello scheduling senza diritto di prelazione, quando si assegna la CPU a un processo, questo rimane in possesso della CPU fino al momento del suo rilascio, dovuto al termine dell'esecuzione o al passaggio nello stato di attesa.

Sfortunatamente lo scheduling con diritto di prelazione presenta un inconveniente. Si consideri il caso in cui due processi condividono dati; mentre uno di questi aggiorna i dati si ha la sua prelazione in favore dell'esecuzione dell'altro. Il secondo processo può, a questo punto, tentare di leggere i dati che sono stati lasciati in uno stato incoerente dal primo processo. Sono quindi necessari nuovi meccanismi per coordinare l'accesso ai dati condivisi; questo argomento è trattato nel Capitolo 6. La capacità di prelazione si ripercuote anche sulla progettazione del kernel del sistema operativo. Durante l'elaborazione di una chiamata di sistema, il kernel può essere impegnato in attività in favore di un processo; tali attività possono comportare la necessità di modifiche a importanti dati del kernel, come le code di I/O. Se si ha la prelazione del processo durante tali modifiche e il kernel (o un driver di dispositivo) deve leggere o modificare gli stessi dati, si può avere il caos.

Dispatcher :

Un altro elemento coinvolto nella funzione di scheduling della CPU è il dispatcher; si tratta del modulo che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine. Questa funzione riguarda quel che segue:

- ◆ il cambio di contesto;
- ◆ il passaggio alla modalità utente;

caratteristiche possono essere utili nella scelta di un appropriato algoritmo di scheduling della CPU.

Scheduler della CPU

Ogni volta la CPU passa nello stato d'inattività, il sistema operativo sceglie per l'esecuzione uno dei processi presenti nella coda dei processi pronti. In particolare, è lo scheduler a breve termine, o scheduler della CPU che, tra i processi in memoria pronti per l'esecuzione, sceglie quello cui assegnare la CPU.

La coda dei processi pronti non è necessariamente una coda in ordine d'arrivo (*first-in, first-out*, FIFO). Come si nota analizzando i diversi algoritmi di scheduling, una coda dei processi pronti si può realizzare come una coda FIFO, una coda con priorità, un albero o semplicemente una lista concatenata non ordinata. Tuttavia, concettualmente tutti i processi della coda dei processi pronti sono posti

- ♦ il salto alla giusta posizione del programma utente per riavviare l'esecuzione.

Poiché si attiva a ogni cambio di contesto, il dispatcher dovrebbe essere quanto più rapido è possibile. Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro è nota come latenza di dispatch.

Criteri di scheduling :

Prima di scegliere l'algoritmo da usare in una specifica situazione occorre considerare le caratteristiche dei diversi algoritmi. Per il confronto tra gli algoritmi di scheduling della CPU sono stati suggeriti molti criteri. Le caratteristiche usate per tale confronto possono incidere in modo rilevante sulla scelta dell'algoritmo migliore.

♦ **Utilizzo della CPU.** La CPU deve essere più attiva possibile. Teoricamente, l'utilizzo della CPU può variare dallo 0 al 100 per cento. In un sistema reale può variare dal 40 per cento, per un sistema con poco carico, al 90 per cento, per un sistema con utilizzo intenso.

♦ **Produttività.** La CPU è attiva quando si svolge del lavoro. Una misura del lavoro svolto è data dal numero dei processi completati nell'unità di tempo: tale misura è detta produttività (*throughput*). Per processi di lunga durata questo rapporto può essere di un processo all'ora, mentre per brevi transazioni è possibile avere una produttività di 10 processi al secondo.

♦ **Tempo di completamento.** Considerando un processo particolare, un criterio importante può essere relativo al tempo necessario per eseguire il processo stesso. L'intervallo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione è chiamato tempo di completamento (*turnaround time*), ed è la somma dei tempi passati nell'attesa dell'ingresso in memoria, nella coda dei processi pronti, durante l'esecuzione nella CPU e nelle operazioni di I/O.

♦ **Tempo d'attesa.** L'algoritmo di scheduling della CPU non influenza sul tempo impiegato per l'esecuzione di un processo o di un'operazione di I/O; influenza solo sul tempo d'attesa nella coda dei processi pronti. Il tempo d'attesa è la somma degli intervalli d'attesa passati nella coda dei processi pronti.

♦ **Tempo di risposta.** In un sistema interattivo il tempo di completamento può non essere il miglior criterio di valutazione: spesso accade che un processo emetta dati abbastanza presto, e continua a calcolare i nuovi risultati mentre quelli precedenti sono in fase d'emissione. Quindi, un'altra misura di confronto è data dal tempo che intercorre tra la sottomissione di una richiesta e la prima risposta prodotta. Questa misura è chiamata tempo di risposta, ed è data dal tempo necessario per iniziare la risposta, ma non dal suo tempo d'emissione. Generalmente il tempo di completamento è limitato dalla velocità del dispositivo d'emissione dei dati.

—>(Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time.
- Min response time

Algoritmi di scheduling :

♦ **Scheduling in ordine d'arrivo (First-come,First-Served→ FCFS)**

Con questo schema la CPU si assegna al processo che la richiede per primo. La realizzazione del criterio FCFS si fonda su una coda FIFO. Quando un processo entra nella coda dei processi pronti, si collega il suo PCB all'ultimo elemento della coda. Quando è libera, si assegna la CPU al processo che si trova alla testa della coda dei processi pronti, rimuovendolo da essa. Il codice per lo scheduling FCFS è semplice sia da scrivere sia da capire. Il tempo medio d'attesa per l'algoritmo FCFS è spesso abbastanza lungo.

ESEMPIO:

Process. Burst Time(millisecondi)

P1	24
P2	3
P3	3

Supponiamo che i processi arrivano in ordine : P1 , P2 , P3



Il diagramma di Gantt(istogramma) per la schedule è:

Tempo di attesa per :

P1 = 0 millisecondi;

P2 = 24 millisecondi;

P3 = 27 millisecondi

Tempo di attesa medio(Average waiting time): $(0 + 24 + 27)/3 = 17$ millisecondi

Se supponiamo che l'ordine è P2,P3,P1:

Avremo il tempo d attesa:

P1=6

P2=0

P3=3

Ed il tempo di attesa medio: $(6+0+3)/3=3$ (migliore rispetto al caso precedente)

Si ha un **effetto convoglio**, tutti i processi attendono che un lungo processo liberi la CPU, che causa una riduzione dell'utilizzo della CPU e dei dispositivi rispetto a quella che si avrebbe se si eseguissero per primi i processi più brevi.

L'algoritmo di scheduling FCFS è senza prelazione; una volta che la CPU è assegnata a un processo, questo la trattiene fino al momento del rilascio, che può essere dovuto al termine dell'esecuzione o alla richiesta di un'operazione di I/O. L'algoritmo FCFS risulta particolarmente problematico nei sistemi a tempo ripartito, dove è importante che ogni utente disponga della CPU a intervalli regolari. Permettere a un solo processo di occupare la CPU per un lungo periodo condurrebbe a risultati disastrosi.

❖ Scheduling per brevità (Shortest-Job-First—>SJF)

Questo algoritmo associa a ogni processo la lunghezza della successiva sequenza di operazioni della CPU. Quando è disponibile, si assegna la CPU al processo che ha la più breve lunghezza della successiva sequenza di operazioni della CPU. Se due processi hanno le successive sequenze di operazioni della CPU della stessa lunghezza si applica lo scheduling FCFS. Lo scheduling si esegue esaminando la lunghezza della successiva sequenza di operazioni della CPU del processo e non la sua lunghezza totale. L'algoritmo SJF può essere sia *con prelazione* sia *senza prelazione*. La scelta si presenta quando alla coda dei processi pronti arriva un nuovo processo mentre un altro processo è ancora in esecuzione. Il nuovo processo può avere una successiva sequenza di operazioni della CPU più breve di quella che resta al processo correntemente in esecuzione. Un algoritmo SJF con prelazione sostituisce il processo attualmente in esecuzione, mentre un algoritmo SJF senza prelazione permette al processo correntemente in esecuzione di portare a termine la propria sequenza di operazioni della CPU.

Due Schemi:

-nonpreemptive= una volta che la CPU è stata assegnata al processo, non può essere prevenuta fino al completamento della sua CPU burst.

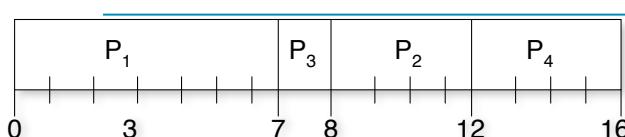
-preemptive= se un nuovo processo arriva con una lunghezza di burst della CPU inferiore al tempo rimanente del processo di esecuzione corrente, effettuare il preempt. Questo schema è noto come Shortest-Remaining-Time-First(SRTF).

ESEMPIO NON-PREEMPTIVE SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

P1	0.0	7	14
P2	2.0	4	12
P3	4.0	1	13
P4	5.0	4	17

- SJF (non-preemptive)



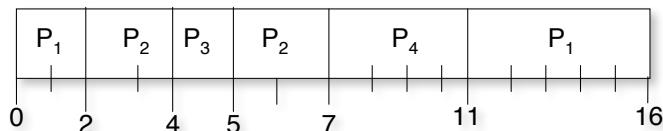
- Tempo medio di attesa=(0+6+3+7)/4=4

ESEMPIO PREEMPTIVE SJF

Process Arrival Time Burst Time

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (preemptive)



- Average waiting time=(9+1+0+2)/4=3

Si può dimostrare che l'algoritmo di scheduling SJF è *ottimale*, nel senso che rende minimo il tempo d'attesa medio per un dato insieme di processi. Spostando un processo breve prima di un processo lungo, il tempo d'attesa per il processo breve diminuisce più di quanto aumenti il tempo d'attesa per il processo lungo. Di conseguenza, il tempo d'attesa *medio* diminuisce. La difficoltà reale implicita nell'algoritmo SJF consiste nel conoscere la durata della successiva richiesta della CPU. Per lo scheduling a lungo termine (*job scheduling*) di un sistema a lotti si può usare come durata il tempo limite d'elaborazione che gli utenti specificano nel sottoporre il processo. Sebbene sia ottimale, l'algoritmo SJF non si può realizzare a livello dello scheduling della CPU a breve termine, poiché non esiste alcun modo per conoscere la lunghezza della successiva sequenza di operazioni della CPU. Un possibile metodo consiste nel tentare di approssimare lo scheduling SJF: se non è possibile *conoscere* la lunghezza della prossima sequenza di operazioni della CPU, si può cercare di *predire* il suo valore; è probabile, infatti, che sia simile ai precedenti. La lunghezza della successiva sequenza di operazioni della CPU generalmente si ottiene calcolando la media esponenziale delle effettive lunghezze delle precedenti sequenze di operazioni della CPU. Denotando con t_n la lunghezza dell' n -esima sequenza di operazioni della CPU e con $T(n+1)$ il valore previsto per la successiva sequenza di operazioni della CPU, con a tale che $0 <= a <= 1$, si definisce ****(T=tau)

$$T(n+1) = a t(n) + (1-a)T(n)$$

Il valore di t_n contiene le informazioni più recenti; T_n registra la storia passata. Il parametro a controlla il peso relativo sulla predizione della storia recente e di quella passata. Se $a = 0$, allora, $T_{n+1} = T_n$, e la storia recente non ha effetto; si suppone, cioè, che le condizioni attuali siano transitorie; se $a = 1$, allora $T_{n+1} = t_n$, e ha significato solo la più recente sequenza di operazioni della CPU: si suppone, cioè, che la storia sia vecchia e irrilevante. Più comune è la condizione in cui $a = 1/2$, valore che indica che la storia recente e la storia passata hanno lo stesso peso.

❖ Scheduling per priorità

L'algoritmo SJF è un caso particolare del più generale algoritmo di scheduling per priorità: si associa una priorità a ogni processo e si assegna la CPU al processo con priorità più alta; i processi con priorità uguali si ordinano secondo uno schema FCFS. Un algoritmo SJF è semplicemente un algoritmo con priorità in cui la priorità (p) è l'inverso della lunghezza (previ sta) della successiva sequenza di operazioni della CPU. A una maggiore lunghezza corrisponde una minore priorità, e viceversa.

ESEMPIO:

Lo scheduling per priorità può essere sia con prelazione sia senza prelazione. Quando un processo arriva alla coda dei processi pronti, si confronta la sua priorità con quella del processo attualmente in esecuzione. Un algoritmo di scheduling per priorità e con diritto di prelazione sottrae la CPU al processo attualmente in esecuzione se la priorità

Processo	Durata della sequenza	Priorità
P_1	10	2

Usando lo scheduling per priorità, questi processi sarebbero ordinati secondo il seguente diagramma di Gantt.

Il tempo d'attesa medio è di 8,2 millisecondi.

dell'ultimo processo arrivato è superiore. Un algoritmo di scheduling senza diritto di prelazione si limita a porre l'ultimo processo arrivato alla testa della coda dei processi pronti. Un problema importante relativo agli algoritmi di scheduling per priorità è l'attesa in definita (**starvation**, letteralmente, "inedia"). Un processo pronto per l'esecuzione, ma che non dispone della CPU, si può considerare bloccato nell'attesa della CPU. Un algoritmo di scheduling per priorità può lasciare processi

con bassa priorità nell'attesa indefinita della CPU. Un flusso costante di processi con priorità maggiore può impedire a un processo con bassa priorità di accedere alla CPU. Una soluzione al problema dell'attesa indefinita dei processi con bassa priorità è costituita dall'invecchiamento (**aging**); si tratta di una tecnica di aumento graduale delle priorità dei processi che attendono nel sistema da parecchio tempo.

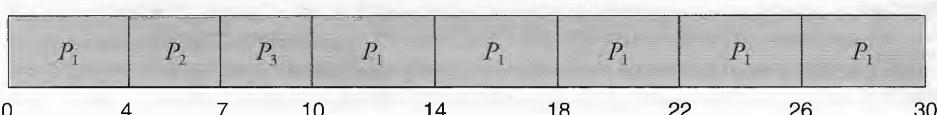
❖ Scheduling circolare (Round-Robin → RR)

L'algoritmo di scheduling circolare (*round-robin*, RR) è stato progettato appositamente per i sistemi a tempo ripartito; simile allo scheduling FCFS, ha tuttavia in più la capacità di prelazione per la commutazione dei processi. Ciascun processo riceve una piccola quantità fissata del tempo della CPU, chiamata quanto di tempo o porzione di tempo (*time slice*), che varia generalmente da 10 a 100 millisecondi; la coda dei processi pronti è trattata come una coda circolare. Lo scheduler della CPU scorre la coda dei processi pronti, assegnando la CPU a ciascun processo per un intervallo di tempo della durata massima di un quanto di tempo. Per realizzare lo scheduling RR si gestisce la coda dei processi pronti come una coda FI FO. I nuovi processi si aggiungono alla fine della coda dei processi pronti. Lo scheduler della CPU individua il primo processo dalla coda dei processi pronti, imposta un timer in modo che invii un segnale d'interruzione alla scadenza di un intervallo pari a un quanto di tempo, e attiva il dispatcher per l'effettiva esecuzione del processo.

A questo punto si può verificare una delle seguenti situazioni: il processo ha una sequenza di operazioni della CPU di durata minore di un quanto di tempo, quindi il processo stesso rilascia volontariamente la CPU e lo scheduler passa al processo successivo della coda dei processi pronti; oppure la durata della sequenza di operazioni della CPU del processo attualmente in esecuzione è più lunga di un quanto di tempo; in questo caso si raggiunge la scadenza del quanto di tempo e il timer invia un segnale d'interruzione al sistema operativo, che esegue un cambio di contesto, aggiunge il processo alla fine della coda dei processi pronti e, tramite lo scheduler della CPU, seleziona il processo successivo nella coda dei processi pronti. Il tempo d'attesa medio per il criterio di scheduling RR è spesso abbastanza lungo.

Esempio:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3



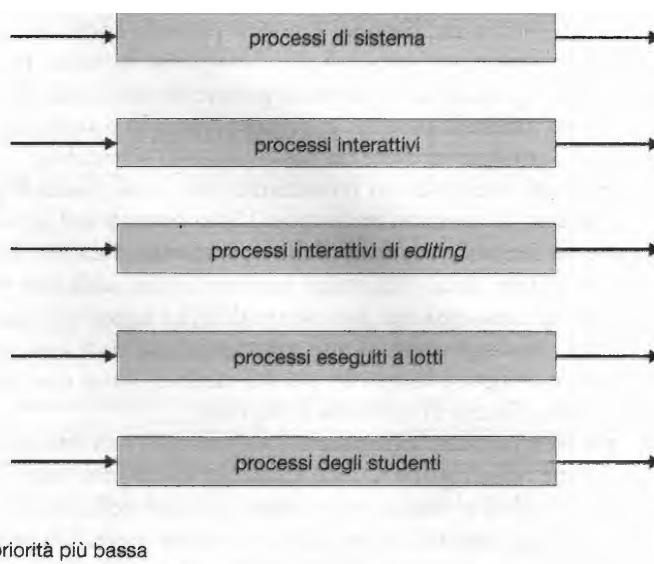
Se si usa un quanto di tempo di 4 millisecondi, il processo P_1 ottiene i primi 4 millisecondi ma, poiché richiede altri 20 millisecondi, è soggetto a prelazione dopo il primo quanto di tempo e la CPU passa al processo successivo della coda, il processo P_2 . Poiché il processo P_2 non necessita di 4 millisecondi, termina prima che il suo quanto di tempo si esaurisca, così si assegna immediatamente la CPU al processo successivo, il processo P_3 . Una volta che tutti i processi hanno ricevuto un quanto di tempo, si assegna nuovamente la CPU al processo P_1 per un ulteriore quanto di tempo.

Calcoliamo ora il tempo di attesa medio per questo scheduling. P_1 resta in attesa per 6 millisecondi ($10-4$), P_2 per 4 millisecondi e P_3 per 7 millisecondi. Il tempo d'attesa medio è di $17/3 = 5,66$ millisecondi.

Nell'algoritmo di scheduling RR la CPU si assegna a un processo per non più di un quanto di tempo per volta. Se la durata della sequenza di operazioni della CPU di un processo eccede il quanto di tempo, il processo viene sottoposto a *prelazione* e riportato nella coda dei processi pronti. L'algoritmo di scheduling RR è pertanto con prelazione. Se nella coda dei processi pronti esistono n processi e il quanto di tempo è pari a q , ciascun processo ottiene $1/n$ -esimo del tempo di elaborazione della CPU in frazioni di, al massimo, q unità di tempo. Ogni processo non deve attendere per più di $(n - 1) \times q$ unità di tempo. Per esempio, dati cinque processi e un quanto di tempo di 20 millisecondi, ogni processo usa 20 millisecondi ogni 100 millisecondi. Le prestazioni dell'algoritmo **RR** dipendono molto dalla dimensione del quanto di tempo. Nel caso limite in cui il quanto di tempo sia molto lungo (indefinito), il criterio di scheduling **RR** si riduce al criterio di scheduling **FCFS**.

❖ Scheduling a code multiple

E stata creata una classe di algoritmi di scheduling adatta a situazioni in cui i processi si possono classificare facilmente in gruppi diversi. Una distinzione diffusa è per esempio quella che si fa tra i processi che si eseguono in primo piano (*foreground*), o interattivi, e i processi che si eseguono in sottofondo (*background*), o a lotti (*batch*). Questi due tipi di processi hanno tempi di risposta diversi e possono quindi avere diverse necessità di scheduling. Inoltre, i processi che si eseguono in primo piano possono avere la priorità, definita esterna mente, sui processi che si eseguono in sottofondo.



5.6 Scheduling a code multiple.

L'algoritmo di scheduling a code multiple (*multilevel queue scheduling algorithm*) suddivide la coda dei processi pronti in code distinte. I processi si assegnano in modo permanente a una coda, generalmente secondo qualche caratteristica del processo, come la quantità di memoria richiesta, la priorità o il tipo. Ogni coda ha il proprio algoritmo di scheduling. Per esempio, per i processi in primo piano e i processi in sottofondo si possono usare code distinte. La coda dei processi in primo piano si può gestire con un algoritmo **RR**, mentre quella dei processi in sottofondo si può gestire con un algoritmo **FCFS**. In questa situazione è inoltre necessario avere uno scheduling tra le code; si tratta comunemente di uno scheduling per priorità fissa e con prelazione. Per esempio, la coda dei processi in primo piano può avere la priorità assoluta sulla coda dei processi in sottofondo.

Si consideri il seguente algoritmo di scheduling a code multiple, in ordine di priorità:

1. processi di sistema;
2. processi interattivi;
3. processi interattivi di *editing*,
4. processi eseguiti in sottofondo;
5. processi degli studenti.

Ogni coda ha la priorità assoluta sulle code di priorità più bassa; nessun processo della coda dei processi in sottofondo può iniziare l'esecuzione finché le code per i processi di sistema, interattivi e interattivi di *editing* non siano tutte vuote. Se un processo interattivo di *editing* entrasse nella coda dei processi pronti durante l'esecuzione di un processo in sottofondo, si avrebbe la prelazione su quest'ultimo. Esiste anche la possibilità di impostare i quanti di tempo per le code. Per ogni coda si stabilisce una parte del tempo d'elaborazione della **CPU**, suddivisibile a sua volta tra i processi che la costituiscono.

Soluzioni di scheduling per multiprocessori

Fin qui la trattazione ha riguardato i problemi inerenti lo scheduling della CPU in un sistema a processore singolo; se sono disponibili più unità d'elaborazione, anche il problema del lo scheduling è proporzionalmente più complesso. Si sono sperimentate diverse possibilità e, come s'è visto nella trattazione dello scheduling di una sola CPU, "la soluzione migliore" non esiste. Si considerano i sistemi in cui le unità d'elaborazione sono, in relazione alle loro funzioni, identiche (**sistemi omogenei**).

Una prima strategia di scheduling della CPU per i sistemi multiprocessore affida tutte le decisioni, l'elaborazione dell'I/O e le altre attività del sistema a un solo processore, il cosiddetto **master server**. Gli altri processori eseguono soltanto il codice dell'utente. Si tratta della **multielaborazione asimmetrica**, che riduce la necessità di condividere dati grazie all'accesso di un solo processore alle strutture dati del sistema.

Quando invece ciascun processore provvede al proprio scheduling, si parla di **multielaborazione simmetrica** (**symmetric multiprocessing**, SMP). In questo caso i processi pronti per l'esecuzione sono situati tutti in una coda comune, oppure vi è un'apposita coda per ogni processore. In entrambi i casi, lo scheduler di ciascun processore esamina la coda appropriata, da cui seleziona un processo da eseguire. Come vedremo nel Capitolo 6, l'accesso concorrente di più processori a una struttura dati comune rende delicata la programmazione dello scheduler, al fine di evitare che due processori scelgano il medesimo processo o che un processo in coda non vada perso. Si parla di **predilezione per il processore (processor affinity)**, intendendo con ciò che un processo ha una predilezione per il processore su cui è in esecuzione. La predilezione per il processore può assumere varie forme. Quando un sistema operativo si propone di mantenere un processo su un singolo processore, ma non garantisce che sarà così, si parla di **predilezione debole (soft affinity)**. In questo caso è possibile che un processo migri da un processore all'altro. Alcuni sistemi, per esempio Linux, dispongono di chiamate di sistema con cui specificare che un processo non debba cambiare processore; in tal modo, si realizza la **predilezione forte (hard affinity)**. Solaris permette che i processi sia no assegnati a un determinato insieme di processori, definendo quali processi possono esse re eseguiti da una data CPU. Solaris permette anche la predilezione debole .

Sommario 5° CAPITOLO

Lo scheduling della CPU consiste nella scelta di un processo dalla coda dei processi pronti cui assegnare la CPU. L'effettiva assegnazione della CPU al processo prescelto è eseguita dal dispatcher.

L'algoritmo di scheduling in ordine d'arrivo (FCFS) è il più semplice, ma può far sì che brevi processi attendano processi molto lunghi. Si dimostra che lo scheduling ottimale, che determina il minimo tempo medio d'attesa, è lo scheduling per brevità (SJF). Realizzare lo scheduling SJF è complicato, poiché è difficile prevedere la lunghezza della successiva sequenza di operazioni della CPU. L'algoritmo SJF è un caso particolare dell'algoritmo generale di scheduling per priorità, che si limita ad assegnare la CPU al processo con priorità più elevata. Sia lo scheduling per priorità sia lo scheduling SJF possono condurre a situazioni d'attesa indefinita. L'invecchiamento (*aging*) è una tecnica che si usa per impedire che avvengano tali situazioni.

Lo scheduling circolare (RR) è il più appropriato per un sistema a tempo ripartito: si assegna la CPU al primo processo della coda dei processi pronti per q unità di tempo (quanto di tempo); dopodiché si ha la prelazione della CPU e si mette il processo in fondo alla coda dei processi pronti. Il problema principale è la scelta della durata del quanto di tempo; se è troppo lungo, lo scheduling RR si riduce a uno scheduling FCFS; se è troppo breve, il carico di scheduling dovuto al tempo dei cambi di contesto diventa eccessivo.

L'algoritmo FCFS è senza prelazione; l'algoritmo RR è con prelazione; gli algoritmi SJF e con priorità possono essere sia con prelazione sia senza prelazione.

Lo scheduling a code multiple permette l'uso di diversi algoritmi per diverse classi di processi. Le più comuni sono la coda dei processi interattivi, eseguiti in primo piano, con scheduling RR, e la coda per processi a lotti, eseguiti in sottofondo, con scheduling FCFS. Le code multiple con retroazione permettono ai processi di spostarsi da una coda all'altra.

Molti dei sistemi attuali supportano processori multipli permettendo a ciascun processore una pianificazione indipendente. In genere ogni processore mantiene una propria coda di processi pronti (o di thread), tutti disponibili all'esecuzione. Tra gli altri aspetti relativi allo scheduling in sistemi multiprocessore vi sono la predilezione, il bilanciamento del carico e lo scheduling con processori multicore e in sistemi di virtualizzazione.

I sistemi operativi che gestiscono i thread a livello kernel devono occuparsi dello scheduling dei thread, e non di quello dei processi. Tra questi vi sono il Solaris e il Windows XP, entrambi gestiscono lo scheduling dei thread impiegando un algoritmo di scheduling con diritto

di prelazione, basato su priorità e che comprende i thread in tempo reale. Anche lo scheduler dei processi di Linux impiega un algoritmo basato su priorità e che prevede la gestione dei processi in tempo reale. Gli algoritmi di scheduling di questi tre sistemi operativi favoriscono generalmente i processi interattivi rispetto ai processi a lotti o con prevalenza d'elaborazione.

La vasta gamma di algoritmi di scheduling esistenti rende imperativa la disponibilità di metodi per la loro selezione. I metodi analitici sfruttano strumenti matematici per valutare le prestazioni degli algoritmi. Le simulazioni determinano le prestazioni eseguendo gli algoritmi in presenza di un insieme “rappresentativo” di processi. Va detto, però, che la simulazione può tutt'al più dare un'approssimazione delle effettive prestazioni dei sistemi. L'unica tecnica affidabile per la valutazione delle prestazioni di un algoritmo di scheduling consiste nell'implementarlo su un vero sistema, e nel misurarne le prestazioni in un contesto reale.

SESTO CAPITOLO

Sincronizzazione dei processi

Un processo cooperante è un processo che può influenzarne un altro in esecuzione nel sistema o anche subirne l'influenza. I processi cooperanti possono condividere direttamente uno spazio logico di indirizzi (cioè, codice e dati) oppure condividere dati soltanto attraverso i file. Nel primo caso si fa uso dei thread, presentati nel Capitolo 4. L'accesso concorrente a dati condivisi può tuttavia causare situazioni di incoerenza degli stessi dati.

Torniamo al concetto di buffer limitato. Come è stato sottolineato, la nostra soluzione consente la presenza contemporanea di non più di `DIM_BUFFER` —1 elementi. Si supponga di voler modificare l'algoritmo per rimediare a questa carenza. Una possibilità consiste nell'aggiungere una variabile intera, `contatore`, inizializzata a 0, che si incrementa ogni volta s'inserisce un nuovo elemento nel buffer e si decrementa ogni qualvolta si preleva un elemento dal buffer. Il codice per il processo produttore si può modificare come segue:

```
while (true)
{
    /* produce un elemento in appena_Prodotto */
    while (contatore == DIM_BUFFER);
    /* non fa niente */
    buffer[inserisci] = appena_Prodotto;
    inserisci = (inserisci +1) % DIM_BUFFER;
    contatore++;
}
```

Il codice per il processo consumatore si può modificare come segue:

```
while (true)
{
    while (contatore == 0)
        /* non fa niente */
    da_Consumare = buffer[preleva];
    preleva = (preleva +1) % DIM_BUFFER;
    contatore--;
    /* consuma un elemento in da_Consumare */
}
```

Sebbene siano corrette se si considerano separatamente, le procedure del produttore e del consumatore possono non funzionare altrettanto correttamente se si eseguono in modo concorrente. Si supponga per esempio che il valore della variabile `contatore` sia attualmente **5**, e che i processi produttore e consumatore eseguano le istruzioni `contatore++` e `contatore--` in modo concorrente. Terminata l'esecuzione delle due istruzioni, il valore della variabile `contatore` potrebbe essere **4, 5 o 6!** Il solo risultato corretto è `contatore == 5`, che si ottiene se si eseguono separatamente il produttore e il consumatore.

Per evitare le situazioni di questo tipo, in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi (le cosiddette **race condition**) occorre assicurare che un solo processo alla volta possa modificare la variabile `contatore`. Questa condizione richiede una forma di sincronizzazione dei processi. Tali situazioni si verificano spesso nei sistemi operativi, nei quali diversi componenti compiono operazioni su risorse condivise. Inoltre, con la diffusione dei sistemi multicore si dà sempre più

importanza allo sviluppo di applicazioni multithread in cui diversi thread, che possono anche condividere dei dati,

Un esempio semplice

- Si consideri il codice seguente:

In C:

```
void modifica(int valore) {  
    totale = totale + valore  
}
```

In Assembly:

```
.text  
modifica:  
    lw $t0, totale  
    add $t0, $t0, $a0  
    sw $t0, totale
```

- Supponiamo che:

- Esista un processo P1 che esegue `modifica(+10)`
- Esista un processo P2 che esegue `modifica(-10)`
- P1 e P2 siano in esecuzione concorrente
- `totale` sia una variabile condivisa tra i due processi, con valore iniziale 100

- Alla fine, `totale` dovrebbe essere uguale a 100. Giusto?

Scenario 2: multiprogramming (errato)

```
P1    lw $t0, totale      totale=100, $t0=100, $a0=10  
S.O.  interrupt  
S.O.  salvataggio registri P1  
S.O.  ripristino registri P2  totale=100, $t0=? , $a0=-10  
P2    lw $t0, totale      totale=100, $t0=100, $a0=-10  
P2    add $t0, $t0, $a0      totale=100, $t0= 90, $a0=-10  
P2    sw $t0, totale      totale= 90, $t0= 90, $a0=-10  
S.O.  interrupt  
S.O.  salvataggio registri P2  
S.O.  ripristino registri P1  totale= 90, $t0=100, $a0=10  
P1    add $t0, $t0, $a0      totale= 90, $t0=110, $a0=10  
P1    sw $t0, totale      totale=110, $t0=110, $a0=10
```

sono in esecuzione in parallelo su unità di calcolo distinte.

Problema della sezione critica

Si consideri un sistema composto di n processi $\{P_0, P_1, \dots, P_{n-1}\}$ ciascuno avente un segmento di codice, chiamato **sezione critica** (detto anche *regione critica*), in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere in un file e così via. Quando un processo è in esecuzione nella propria sezione critica, non si deve consentire a nessun altro processo di essere in esecuzione nella propria sezione critica. Quindi,

l'esecuzione delle sezioni critiche da parte dei processi è *mutuamente esclusiva* nel tempo. Il problema della **sezione critica** si affronta progettando un protocollo che i processi possono usare per cooperare. Ogni processo deve chiedere il permesso per entrare nella propria sezione critica. La sezione di codice che realizza questa richiesta è la **sezione d'ingresso**. La sezione critica può essere seguita da una **sezione d'uscita**, e la restante parte del codice è detta **sezione non critica**.

```
do{  
    sezione d'ingresso  
    sezione critica  
    sezione d'uscita  
    sezione non critica  
} while (true);
```

Struttura generale di un tipico processo P_i .

Scenario 1: multiprogramming (corretto)

```
P1    lw $t0, totale      totale=100, $t0=100, $a0=10  
P1    add $t0, $t0, $a0      totale=100, $t0=110, $a0=10  
P1    sw $t0, totale      totale=110, $t0=110, $a0=10  
S.O.  interrupt  
S.O.  salvataggio registri P1  
S.O.  ripristino registri P2  totale=110, $t0=? , $a0=-10  
P2    lw $t0, totale      totale=110, $t0=110, $a0=-10  
P2    add $t0, $t0, $a0      totale=110, $t0=100, $a0=-10  
P2    sw $t0, totale      totale=100, $t0=100, $a0=-10
```

Scenario 3: multiprocessing (errato)

- Il due processi vengono eseguiti simultaneamente da due processori distinti

Processo P1: <code>lw \$t0, totale</code> <code>add \$t0, \$t0, \$a0</code> <code>sw \$t0, totale</code>	Processo P2: <code>lw \$t0, totale</code> <code>add \$t0, \$t0, \$a0</code> <code>sw \$t0, totale</code>
--	--

- Nota:
 - i due processi hanno insiemi di registri distinti
 - l'accesso alla memoria su `totale` non può essere simultaneo

Una soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti:

- Mutua esclusione.** Se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
- Progresso.** Se nessun processo è in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori delle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non si può rimandare indefinitamente.
- Attesa limitata.** Se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

Le due strategie principali per la gestione delle sezioni critiche nei sistemi operativi prevedono l'impiego di: (1) **kernel con diritto di prelazione** e (2) **kernel senza diritto di prelazione**. Un kernel con diritto di prelazione consente che un processo funzionante in modalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l'esecuzione. Un kernel senza diritto di prelazione non consente di applicare la prelazione a un processo attivo in modalità di sistema: l'esecuzione di questo processo seguirà finché lo stesso esca da tale modalità, si blocchi o ceda volontariamente il controllo della CPU. In sostanza, i kernel senza diritto di prelazione sono immuni dai problemi legati all'ordine degli accessi alle strutture dati del kernel, visto che un solo processo per volta impegna il kernel. Altrettanto non si può dire dei kernel con diritto di prelazione, motivo per cui bisogna avere cura, nella progettazione, di mantenerli al riparo dai problemi insiti nell'ordine degli accessi. I kernel con diritto di prelazione sono più adatti alla programmazione real-time, dal momento che permette ai processi in tempo reale di far valere il loro diritto di precedenza nei confronti di un processo attivo nel kernel. Inoltre, i kernel con diritto di prelazione possono vantare una maggiore prontezza nelle risposte, data la loro scarsa propensione a eseguire i processi in modalità di sistema per un tempo eccessivamente lungo, prima di liberare la CPU per i processi in attesa. Naturalmente, il fatto che questo effetto sia notevole o trascurabile dipende dai dettagli del codice del kernel. Vedremo più avanti come diversi sistemi operativi usino la prelazione all'interno del kernel.

Soluzione di Peterson

La soluzione di Peterson si applica a due processi, P_0 e P_1 ognuno dei quali esegue alternativamente la propria sezione critica e la sezione rimanente. Per il seguito, è utile convenire che se P_i denota uno dei due processi, P_j denoti l'altro; ossia, che $j = 1 - i$.

La soluzione di Peterson richiede che i processi condividano i seguenti dati:

int turno;

boolean flag[2];

La variabile turno segnala, per l'appunto, di chi sia il turno d'accesso alla sezione critica; quindi, se $\text{turno} == i$, il processo P_i è autorizzato a eseguire la propria sezione critica. L'array flag, invece, indica se un processo sia pronto a entrare nella propria sezione critica. Per esempio, se $\text{flag}[i]$ è true, P_i lo è.

```

do {
    flag[i] = true;
    turno = j;
    while (flag[j] && turno == j);

    sezione critica

    flag[i] = false;

    sezione non critica
} while (true);

```

.2 Struttura del processo P_i nella soluzione di Peterson.

Per accedere alla sezione critica, il processo P_i assegna innanzitutto a $\text{flag}[i]$ il valore true; quindi attribuisce a turno il valore j , conferendo così all'altro processo la facoltà di entrare nella sezione critica. Qualora entrambi i processi tentino l'accesso contemporaneo, all'incirca nello stesso momento sarà assegnato a turno sia il valore i sia il valore j . Soltanto uno dei due permane: l'altro sarà immediatamente sovrascritto. Il valore definitivo di turno stabilisce quale dei due processi sia autorizzato a entrare per primo nella propria sezione critica. Dimostriamo ora la correttezza di questa soluzione. Dobbiamo provare che:

- la mutua esclusione è preservata;
- il requisito del progresso è soddisfatto;

- il requisito dell'attesa limitata è rispettato.

Per dimostrare la proprietà 1, si osservi come ogni P_i acceda alla propria sezione critica solo se $\text{flag}[j] == \text{false}$ oppure $\text{turno} == i$. Si noti anche che, se entrambi i processi sono eseguibili in concomitanza nelle rispettive sezioni critiche, allora $\text{flag}[0] == \text{flag}[1] == \text{true}$. Si desume da queste due osservazioni che P_0 e P_1 sono impossibilitati a eseguire con successo le rispettive istruzioni $w h i l e$ approssimativamente nello stesso momento: turno, infatti, può valere 0 o 1, ma non entrambi. Pertanto, uno dei processi - poniamo P_i — deve aver eseguito con successo l'istruzione $w h i l e$, mentre P_j aveva da eseguire almeno un'istruzione aggiuntiva (“turno == j”). Tuttavia, poiché da quel momento, e fino al termine della permanenza di P_j nella propria sezione critica, restano valide le asserzioni $\text{flag}[j] == \text{true}$ e $\text{turno} == j$, ne consegue che la mutua esclusione è preservata.

Per dimostrare le proprietà 2 e 3, osserviamo come l'ingresso di un processo P_i nella propria sezione critica possa essere impedito solo se il processo è bloccato nella sua iterazione while, con le condizioni $\text{flag}[j] == \text{true}$ e $\text{turno} == j$; questa è l'unica possibilità. Qualora P_j , non sia pronto a entrare nella sezione critica, $\text{flag}[j] == \text{false}$, e P_i può accedere alla propria sezione critica. Se P_j ha impostato $f l a g [j] = t r u e$ e sta eseguendo il proprio ciclo while, $\text{turno} == i$, oppure $\text{turno} == j$. Se $\text{turno} == i$, P_i entrerà nel la propria sezione critica. Se $\text{turno} == j$, P_j entrerà nella propria sezione critica. Tuttavia, al momento di uscire dalla propria sezione critica, P_j reimposta $f l a g [j] = f a l s e$, consentendo a P_i di entrarvi. Se P_j imposta $\text{flag}[j]$ a true, deve anche attribuire alla variabile turno il valore i. Poiché tuttavia P_i non modifica il valore della variabile turno durante l'esecuzione dell'istruzione while, P_i entrerà nella sezione critica (progresso) dopo che P_j abbia effettuato non più di un ingresso (attesa limitata).

Hardware per la sincronizzazione

```
do{
    acquisisce il lock
    sezione critica
    restituisce il lock
    sezione non critica
} while (true);
```

Figura 6.3 Soluzione al problema della sezione critica tramite lock.

```
do {
    while (TestAndSet(&lock));
    sezione critica
    lock = false;
    sezione non critica
} while (true);
```

Figura 6.5 Realizzazione di mutua esclusione con `TestAndSet()`.

```
do {
    chiave = true;
    while (chiave == true)
        Swap(&lock, &chiave);
    sezione critica
    lock = false;
    sezione non critica
} while (true);
```

Figura 6.7 Realizzazione di mutua esclusione con `Swap()`.

In generale, si può affermare che qualunque soluzione al problema richiede l'uso di un semplice strumento detto lock (*lucchetto*). Il corretto ordine degli accessi alle strutture dati del kernel è garantito dal fatto che le sezioni critiche sono protette da lock. In altri termini, per accedere alla propria sezione critica un processo deve acquisire il possesso di un lock, che restituirà al momento della sua uscita.

Molte delle moderne architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria, oppure di scambiare il contenuto di due parole di memoria, in modo **atomico** – cioè come un'unità non interrompibile. Queste speciali istruzioni sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice. Anziché discutere una specifica istruzione di una particolare architettura, è preferibile astrarre i concetti principali che stanno alla base di queste istruzioni.

L'istruzione `TestAndSet()` si può definire com'è illustrato nella Figura 6.4. Questa istruzione è eseguita **atomicamente**, cioè come un'unità non soggetta a interruzioni; quindi, se si eseguono contemporaneamente due istruzioni `TestAndSet()`, ciascuna in un'unità d'elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario. L'istruzione `Swap()`, definita nella Figura 6.6, agisce sul contenuto di due parole di memoria; come l'istruzione `TestAndSet()`, è anch'essa eseguita atomicamente. Se si dispone dell'istruzione `Swap()`, la mutua esclusione si garantisce dichiarando e inizializzando al valore **false** una variabile booleana globale `lock`. Inoltre, ogni processo possiede anche una variabile booleana locale `chiave`. La struttura del processo P_t è illustrata nella Figura

Semafori

Un **semaforo S** è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: `wait()` e `signal()`.

La definizione classica di `wait()` in pseudocodice è la seguente:

```
wait(S) {  
    while(S <= 0)  
        ;//non-op  
    S --;  
}
```

La definizione classica di `signal()` in pseudocodice è la seguente:

```
signal(S) {  
    S++;  
}
```

Tutte le modifiche al valore del semaforo contenute nelle operazioni `wait()` e `signal()` si devono eseguire in modo indivisibile: mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore. Inoltre, nel caso della `wait(S)` si devono eseguire senza interruzione anche la verifica del valore intero di `S (S < 0)` e la sua possibile modifica (`S --`).

Uso dei semafori

Si usa distinguere tra semafori contatore, il cui valore numerico è illimitato, e i semafori binari, il cui valore è 0 o 1. In relazione a certi sistemi i semafori binari sono anche detti lock mutex (*mutex locks*), perché fungono da “lock” che garantiscono la mutua esclusione (dall'inglese mutual exclusion).

I semafori sono utilizzabili per risolvere il problema della sezione critica con n processi. Gli n processi condividono un semaforo comune, mutex, inizializzato a 1. Ogni processo P_i è strutturato com'è illustrato nella Figura 6.9.

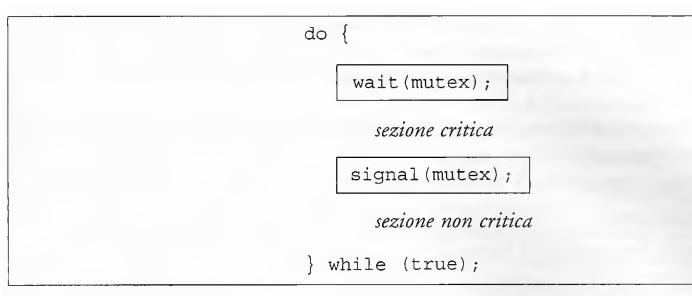


Figura 6.9 Realizzazione di mutua esclusione con semafori.

I processi che desiderino utilizzare un esemplare della risorsa invocano `wait()` sul semaforo, decrementandone così il valore; i processi che restituiscono un esemplare della risorsa, invece, invocano `signal()` sul semaforo, incrementandone il valore. Quando il semaforo vale 0, vengono allocati tutti gli esemplari della risorsa, e i processi che ne richiedano l'uso dovranno bloccarsi fino a che il semaforo non ritorni positivo. I semafori sono utilizzabili anche per risolvere diversi problemi di sincronizzazione.

Si considerino, per esempio, due processi in esecuzione concorrente: P_1 con un'istruzione `S1` e P_2 con un'istruzione `S2`. Si supponga di voler eseguire `S2` solo dopo che `S1` è terminata.

Questo schema si può prontamente realizzare facendo condividere a P_1 e P_2 un semaforo comune, sincronizzazione, inizializzato a 0, e inserendo nel processo P_1 le istruzioni

```
S1;  
signal(sincronizzazione);
```

```
e nel processo P2 le istruzioni  
wait(sincronizzazione);  
S2?;
```

Poiché sincronizzazione è inizializzato a 0, P_2 esegue `S2` solo dopo che P_1 ha eseguito `signal(sincronizzazione)`, che si trova dopo `S1`.

Il principale svantaggio della definizione di semaforo è che richiede una condizione di **attesa attiva** (*busy waiting*). Mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrarvi si trova sempre nel ciclo del codice della sezione d'ingresso. Chiaramente questa soluzione costituisce un problema per un sistema con multiprogrammazione, poiché la condizione d'attesa attiva spreca cicli della CPU che un altro

processo potrebbe sfruttare in modo produttivo. Questo tipo di semaforo è anche detto **spinlock**, perché i processi “girano” (*spin*) mentre attendono al semaforo. (I semafori spinlock hanno però il vantaggio di non richiedere cambio di contesto nel caso in cui un processo sia fermo in attesa. Tali cambi di contesto possono essere piuttosto costosi, in termini di tempo. Ne consegue che i semafori spinlock sono utili quando i lock sono applicati per brevi intervalli di tempo: infatti, trovano frequente applicazione nei sistemi multiprocessore, dove un processo gira su un processore mentre un altro thread esegue la propria sezione critica su un altro processore.)

Per superare la necessità dell’attesa attiva, si possono modificare le definizioni delle operazioni `wait()` e `signal()`: quando un processo invoca l’operazione `wait()` e trova che il valore del semaforo non è positivo, deve attendere, ma anziché restare nell’attesa attiva può *bloccare* se stesso. L’operazione di *bloccaggio* pone il processo in una coda d’attesa associata al semaforo e cambia lo stato del processo nello stato d’attesa.

Un processo bloccato, che attende a un semaforo **S**, sarà riavviato in seguito all’esecuzione di un’operazione `sigнал()` su **S** da parte di qualche altro processo. Il processo si riavvia tramite un’operazione `wakeup()`, che

Per realizzare i semafori secondo quel che s’è detto si può definire il semaforo come una struttura del linguaggio C:

```
typedef struct {
    int valore;
    struct processo *lista;
} semaforo;
```

A ogni semaforo sono associati un valore intero e una lista di processi, contenente i processi in attesa a un semaforo; l’operazione `signal()` preleva un processo da tale lista e lo attiva.

L’operazione `wait()` del semaforo si può definire come segue:

```
wait(semaforo *S) {
    S->valore--;
    if (S->valore < 0) {
        aggiungi questo processo a S->lista;
        block();
    }
}
```

L’operazione `signal()` del semaforo si può definire come segue:

```
signal(semaforo *S) {
    S->valore++;
    if (S->valore <= 0) {
        togli un processo P da S->lista;
        wakeup(P);
    }
}
```

list di PCB.

I semafori devono essere eseguiti in modo atomico. Si deve garantire che nessuno dei due processi possa eseguire operazioni `wait()` e `signal()` contemporaneamente sullo stesso semaforo. Si tratta di un problema di accesso alla sezione critica, e in un contesto monoprocesso si può risolvere semplicemente inibendo le interruzioni durante l’esecuzione di `signal()` e `wait()`. Nei sistemi con una sola CPU, infatti, le interruzioni sono i soli elementi di disturbo: non vi sono istruzioni eseguite da altri processori. Finché non si riattivino le interruzioni, dando la possibilità allo scheduler di riprendere il controllo della CPU, il processo corrente continua indisturbato la sua esecuzione. Nei sistemi multiprocessore è necessario disabilitare le interruzioni di tutti i processori, perché altrimenti le istruzioni dei diversi processi in esecuzione su processori distinti potrebbero interferire fra loro.

È importante rilevare che questa definizione delle operazioni `wait()` e `signal()` non consente di eliminare completamente l’attesa attiva, ma piuttosto di rimuoverla dalle sezioni d’ingresso dei programmi applicativi.

Inoltre, l’attesa attiva si limita alle sezioni critiche delle operazioni `wait()` e `signal()`, che sono abbastanza brevi; se sono convenientemente codificate, non sono più lunghe di 10 istruzioni. Quindi, la sezione critica non è quasi mai occupata e l’attesa attiva si presenta raramente e per breve tempo. Una situazione completamente diversa si verifica con i programmi applicativi le cui sezioni critiche possono essere lunghe minuti o anche ore, oppure occupate spesso. In questi casi l’attesa attiva è assai inefficiente.

Stallo/Deadlocked e attesa indefinita

La realizzazione di un semaforo con coda d’attesa può condurre a situazioni in cui ciascun processo di un insieme di processi attende indefinitamente un evento —l’esecuzione di un’operazione `signal()`—che può essere causato solo da uno dei processi dello stesso insieme. Quando si verifica una situazione di questo tipo si dice che i processi sono in stallo (*deadlocked*).

modifica lo stato del processo da attesa a pronto. Il processo entra nella coda dei processi pronti.

L’operazione `block()` sospende il processo che la invoca; l’operazione `wakeup(P)` pone in stato di pronto per l’esecuzione un processo **P** bloccato. Queste due operazioni sono fornite dal sistema operativo come chiamate di sistema di base.

Occorre notare che, mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, tale definizione può condurre a valori negativi *vi*. Se il valore del semaforo è negativo, la sua dimensione è data dal numero dei processi che attendono a quel semaforo. Ciò avviene a causa dell’inversione dell’ordine del decremento e della verifica nel codice dell’operazione `wait()`.

La lista dei processi che attendono a un semaforo si può facilmente realizzare inserendo un campo puntatore in ciascun blocco di controllo del processo (PCB). Ogni semaforo contiene un valore intero e un puntatore a una

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

Per illustrare questo fenomeno si consideri un insieme di due processi, P_0 e P_1 ciascuno dei quali ha accesso a due semafori, S e Q , impostati al valore 1: Si supponga che P_0 esegua wait (S) e quindi P_1 esegua wait (Q); eseguita wait (Q), P_0 deve attendere che P_1 esegua signal (Q); analogamente, quando P_1 esegue wait (S), deve attendere che P_0 esegua signal (S). Poiché queste operazioni signal () non si possono eseguire, P_0 e P_1 sono in stallo.

Un insieme di processi è in stallo se ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme. In questo contesto si considerano principalmente gli

eventi di acquisizione e rilascio di risorse, tuttavia anche altri tipi di eventi possono produrre situazioni di stallo (si veda il Capitolo 7, che descrive anche i meccanismi che servono ad affrontare questo tipo di problema).

Un'altra questione connessa alle situazioni di stallo è quella dell'attesa indefinita (nota anche col termine *starvation*). Con tale termine si definisce una situazione d'attesa indefinita nella coda di un semaforo, che si può per esempio presentare se i processi si aggiungono e si rimuovono dalla lista associata a un semaforo secondo un criterio LIFO (last-in, first-out).

Inversione di priorità

Nello scheduling dei processi si possono incontrare difficoltà ogniqualvolta un processo a priorità più alta abbia bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa. Visto che i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse. La situazione si complica ulteriormente se il processo a priorità più bassa ha dovere di prelazione su un processo a priorità più alta. Questo problema è noto come **inversione della priorità**. Dato che l'inversione di priorità si verifica solo su sistemi con più di due priorità, una delle soluzioni è limitare a due il numero di priorità. Tuttavia, questa soluzione non è accettabile nella maggior parte dei sistemi a uso generale. Solitamente questi sistemi risolvono il problema implementando un **protocollo di ereditarietà delle priorità**, secondo il quale tutti i processi che stanno accedendo a risorse di cui hanno bisogno processi con priorità maggiore ereditano la priorità più alta finché non finiscono di utilizzare le risorse in questione. Quando hanno terminato, la loro priorità ritorna al valore originale.

Problemi tipici di sincronizzazione

Produttori e consumatori con memoria limitata :

Il problema dei *produttori e consumatori con memoria limitata*, trattato anche nel Paragrafo 6.1, si usa generalmente per illustrare la potenza delle primitive di sincronizzazione. In questa sede si presenta uno schema generale di soluzione, senza far riferimento a nessuna realizzazione particolare. In conclusione del capitolo proponiamo al riguardo un progetto di programmazione.

```

do {
    produce un elemento in appena_Prodotto
    .
    .
    .
    wait(vuote);
    wait(mutex);
    inserisci in buffer l'elemento in appena_Prodotto
    .
    .
    .
    signal(mutex);
    signal(piene);
} while (true);

```

```

do {
    wait(piene);
    wait(mutex);
    rimuovi un elemento da buffer e mettilo in da_Consumare
    .
    .
    .
    signal(mutex);
    signal(vuote);
    .
    .
    .
    consuma l'elemento contenuto in da_Consumare
} while (true);

```

Si supponga di disporre di una certa quantità di memoria rappresentata da un buffer con n posizioni, ciascuna capace di contenere un elemento. Il semaforo mutex garantisce la mutua esclusione degli accessi al buffer ed è inizializzato al valore 1. I semafori vuote e piene conteggiano rispettivamente il numero di posizioni vuote e il numero di posizioni piene nel buffer. Il semaforo vuote si inizializza al valore n ; il semaforo piene si inizializza al valore 0.

La Figura 6.10 riporta la struttura generale del processo produttore, la Figura 6.11 quella del processo consumatore. È interessante notare la simmetria esistente tra il produttore e il consumatore. Il codice si può interpretare nel senso di produzione, da parte del

Figura 6.11 Struttura generale del processo consumatore.

produttore, di posizioni piene per il consumatore; oppure di produzione, da parte del consumatore, di posizioni vuote per il produttore.

Problema dei lettori-scrittori

Si supponga che una base di dati da condividere tra numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura del contenuto dell'oggetto condiviso, mentre altri possono richiedere un aggiornamento, vale a dire una lettura e una scrittura, dello stesso oggetto. Questi due processi sono distinti, e si indicano chiamando **lettori** quelli interessati al la sola lettura e **scrittori** gli altri. Naturalmente, se due lettori accedono nello stesso momento all'insieme di dati condiviso, non si ha alcun effetto negativo; viceversa, se uno scrittore e un altro processo (lettore o scrittore) accedono contemporaneamente alla stessa base di dati, ne può derivare il caos. Per impedire l'insorgere di difficoltà di questo tipo è necessario che gli scrittori abbia no un accesso esclusivo alla base di dati condivisa. Questo problema di sincronizzazione è conosciuto come **problema dei lettori-scrittori**. Da quando tale problema fu enunciato, è stato usato per verificare quasi tutte le nuove primitive di sincronizzazione. Il problema dei lettori-scrittori ha diverse varianti, che implicano tutte l'esistenza di priorità; la più semplice, cui si fa riferimento come al *primo* problema dei lettori-scrittori, richiede che nessun lettore attenda, a meno che uno scrittore abbia già ottenuto il permesso di usare l'insieme di dati condiviso. In altre parole, nessun lettore deve attendere che altri lettori terminino l'operazione solo perché uno scrittore attende l'accesso ai dati. Il *secondo* problema dei lettori- scrittori si fonda sul presupposto che uno scrittore, una volta pronto, esegua il proprio compito di scrittura al più presto. In altre parole, se uno scrittore attende l'accesso all'insieme di dati, nessun nuovo lettore deve iniziare la lettura.

```
do {  
    wait(scrittura);  
    . . .  
    esegui l'operazione di scrittura  
    . . .  
    signal(scrittura);  
}while (true);
```

Figura 6.12 Struttura generale di un processo scrittore.

```
do {  
    wait(mutex);  
    numlettori++;  
    if (numlettori == 1)  
        wait(scrittura);  
    signal(mutex);  
    . . .  
    esegui l'operazione di lettura  
    . . .  
    wait(mutex);  
    numlettori--;  
    if (numlettori == 0)  
        signal(mutex);  
}while (true);
```

La soluzione del primo problema e quella del secondo possono condurre a uno stato d'attesa indefinita (*starvation*), degli scrittori, nel primo caso; dei lettori, nel secondo. Per questo motivo sono state proposte altre varianti. La Figura 6.12 illustra la struttura generale di un processo scrittore; la Figura 6.13 pre senta la struttura generale di un processo lettore. Occorre notare che se uno scrittore si trova nella sezione critica e n lettori attendono di entrarvi, si accoda un lettore a s c r i t t u r a e $n-1$ lettori a mutex. Inoltre, se uno scrittore esegue signal(scrittura)si può riprendere l'esecuzione dei lettori in attesa, oppure di un singolo scrittore in attesa. La scelta è fatta dallo scheduler.

Le soluzioni al problema dei lettori-scrittori sono state generalizzate su alcuni sistemi in modo da fornire **lock di lettura-scrittura**. Per acquisire un tale lock, è necessario specificare la modalità di scrittura o di lettura: se il processo desidera solo leggere i dati condivisi, richiede un lock di lettura-scrittura in modalità lettura; se invece desidera anche modificare i dati, lo richiede in modalità scrittura. E permesso a più processi di acquisire lock di lettura-scrittura in modalità lettura, ma solo un processo alla volta può avere il lock di lettura- scrittura in modalità scrittura, visto che nel caso della scrittura è necessario garantire l'accesso esclusivo.

Figura 6.13 Struttura generale di un processo lettore.

Sommario 6° CAPITOLO

Dato un gruppo di processi sequenziali cooperanti che condividono dati, è necessario garantirne la mutua esclusione. Si tratta di assicurare che una sezione critica di codice sia utilizzabile da un solo processo o thread alla volta. Di solito l'hardware di un calcolatore fornisce diverse operazioni che assicurano la mutua esclusione, ma per la maggior parte dei programmati questi soluzioni hardware sono troppo complicate da utilizzare. I semafori rappresentano una soluzione a questo problema. I semafori consentono di superare questa difficoltà; sono utilizzabili per risolvere diversi problemi di sincronizzazione e sono realizzabili in modo efficiente, soprattutto se è disponibile un'architettura che permette le operazioni atomiche.

Sono stati presentati diversi problemi di sincronizzazione - come il problema dei produttori e consumatori con memoria limitata, dei lettori-scrittori e dei cinque filosofi - che costituiscono esempi rappresentativi di una vasta classe di problemi di controllo della concorrenza. Questi problemi sono stati usati per verificare quasi tutti gli schemi di sincronizzazione proposti.

Il sistema operativo deve fornire mezzi di protezione contro gli errori di sincronizzazione. A tal fine sono stati proposti parecchi costrutti di linguaggio. I monitor offrono un meccanismo di sincronizzazione per la condivisione di tipi di dati astratti. Una variabile condizionale consente a una procedura di monitor di sospendere la propria esecuzione finché non riceve un segnale.

Solaris, Windows XP e Linux sono esempi di sistemi operativi moderni che offrono vari meccanismi come semafori, mutex, spinlock e variabili condizionali per il controllo del l'accesso ai dati condivisi. La API Pthreads fornisce supporto a mutex e variabili condizionali.

Una transazione è un'unità di programma da eseguire in modo atomico, cioè le operazioni a essa associate vanno eseguite nella loro totalità o non si devono eseguire per niente. Per assicurare la proprietà di atomicità anche nel caso di malfunzionamenti, si può usare la registrazione con scrittura anticipata. Si registrano tutti gli aggiornamenti nel log, che si mantiene in una memoria stabile. Se si verifica un crollo del sistema, si usano le informazioni conservate nel log per ripristinare lo stato dei dati aggiornati; tale ripristino si esegue usando le operazioni undo e redo. Per ridurre il carico dovuto alla ricerca nel log dopo un malfunzionamento del sistema è utilizzabile un metodo basato su punti di verifica.

Per assicurare una corretta esecuzione si deve usare uno schema di controllo della concorrenza che garantisca la serializzabilità. Esistono diversi schemi di controllo della concorrenza che assicurano la serializzabilità differendo un'operazione o arrestando la transazione che ha richiesto l'operazione. I più diffusi sono i protocolli per la gestione dei lock e gli schemi d'ordinamento a marche temporali.

OTTAVO CAPITOLO

MEMORIA CENTRALE

Un tipico ciclo d'esecuzione di un'istruzione, per esempio, prevede che l'istruzione sia prelevata dalla memoria; decodificata (e ciò può comportare il prelievo di operandi dalla memoria) ed eseguita sugli eventuali operandi; i risultati si possono salvare in memoria. La memoria *vede* soltanto un flusso d'indirizzi di memoria, e non *sa* come sono generati oppure a che cosa servano (istruzioni o dati). Di conseguenza, è possibile ignorare *come* un programma genera un indirizzo di memoria, e prestare attenzione solo alla sequenza degli indirizzi di memoria generati dal programma in esecuzione. L'analisi che sviluppiamo nel seguito comprende una panoramica degli aspetti basilari dell'architettura, dagli spazi di indirizzi logici e fisici agli indirizzi fisici reali, offrendo una loro analisi comparativa. La memoria centrale e i registri incorporati nel processore sono le sole aree di memorizzazione a cui la CPU può accedere direttamente. Qualsiasi istruzione in esecuzione, e tutti i dati utilizzati dalle istruzioni, devono risiedere in uno di questi dispositivi per la memorizzazione ad accesso diretto. I dati che non sono in memoria devono essere caricati prima che la CPU possa operare su di loro. Nei casi in cui l'accesso alla memoria richieda molti cicli d'orologio, il processore entra necessariamente in stallo (*stall*), poiché manca dei dati richiesti per completare l'istruzione che sta eseguendo. Questa situazione è intollerabile, perché gli accessi alla memoria sono frequenti. Il rimedio consiste nell'interposizione di una memoria veloce tra CPU e memoria centrale. Un buffer di memoria, detto *cache*, è in grado di conciliare le differenti velocità. Innanzitutto, bisogna assicurarsi che ciascun processo abbia uno spazio di memoria separato. A tal fine, occorre poter determinare l'intervallo degli indirizzi a cui un processo può accedere legalmente, e garantire che possa accedere soltanto a questi indirizzi. Si può implementare il meccanismo di protezione tramite due registri, detti **registri base** e **registri limite**, come

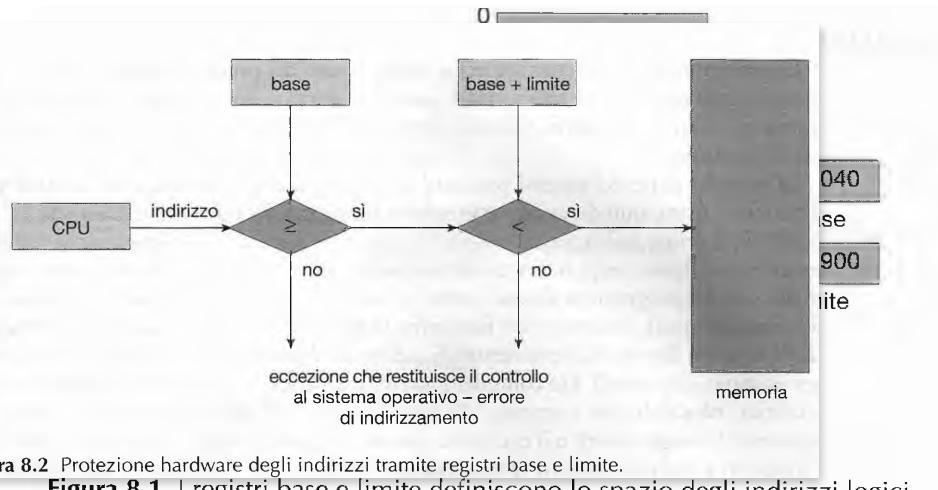


Figura 8.2 Protezione hardware degli indirizzi tramite registri base e limite.

Figura 8.1 I registri base e limite definiscono lo spazio degli indirizzi logici.

illustrato nella Figura 8.1. Il registro base contiene il più piccolo indirizzo legale della memoria fisica; il registro limite determina la dimensione dell'intervallo ammesso. Ad esempio, se i registri base e limite contengono rispettivamente i valori 300040 e 120900, al programma si consente l'accesso alle locazioni di memoria di indirizzi compresi tra 300040 e 420939, estremi inclusi.

Per mettere in atto il meccanismo di protezione, la CPU confronta *ciascun* indirizzo generato in modalità utente con i valori contenuti nei due registri. Qualsiasi

tentativo da parte di un programma eseguito in modalità utente di accedere alle aree di memoria riservate al sistema operativo o a una qualsiasi area di memoria riservata ad altri utenti comporta l'invio di un segnale di eccezione che restituisce il controllo al sistema operativo che, a sua volta, interpreta l'evento come un errore fatale (Figura 8.2). Questo schema impedisce a qualsiasi programma utente di alterare (accidentalmente o intenzionalmente) il codice o le strutture dati, sia del sistema operativo sia degli altri utenti.

Associazione degli indirizzi

In genere un programma risiede in un disco in forma di un file binario eseguibile. Per essere eseguito, il programma va caricato in memoria e inserito all'interno di un processo. Secondo il tipo di gestione della memoria adoperato, durante la sua esecuzione, il processo si può trasferire dalla memoria al disco e viceversa. L'insieme dei processi presenti nei dischi e che attendono d'essere trasferiti in memoria per essere eseguiti forma la coda d'ingresso (*input queue*). La procedura normale consiste nello scegliere uno dei processi appartenenti alla coda d'ingresso e nel caricarlo in memoria. Il processo durante l'esecuzione può accedere alle istruzioni e ai dati in memoria. Quando il processo termina, si dichiara disponibile il suo spazio di memoria. Nella maggior parte dei casi un programma utente, prima di essere eseguito, deve passare attraverso vari stadi, alcuni dei quali possono essere facoltativi (Figura 8.3), in cui gli indirizzi sono rappresentabili in modi diversi.

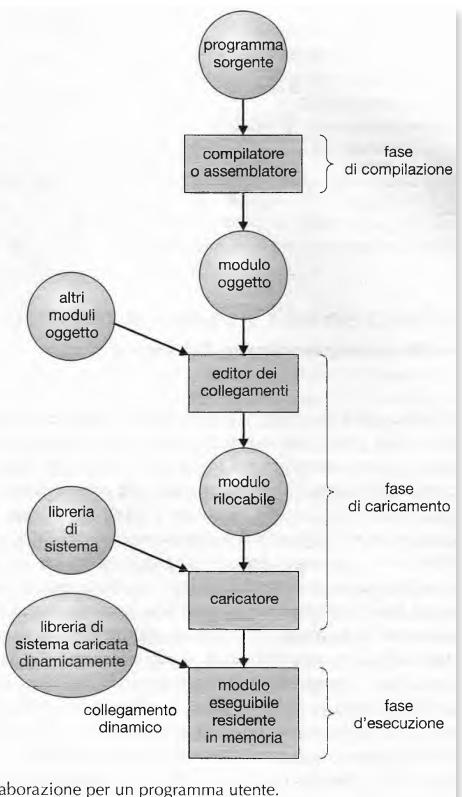


Figura 8.3 Fasi di elaborazione per un programma utente.

Generalmente, l'associazione di istruzioni e dati a indirizzi di memoria si può compiere in qualsiasi fase del seguente percorso:

♦ **Compilazione.** Se nella fase di compilazione si sa dove il processo risiederà in memoria, si può generare **codice assoluto**. Se, per esempio, è noto a priori che un processo utente inizia alla locazione *r*, anche il codice generato dal compilatore comincia da quella locazione. Se, in un momento successivo, la locazione iniziale cambiasse, sarebbe necessario ricompilare il codice. I programmi per MS-DOS nel formato identificato dall'estensione .COM sono collegati al tempo di compilazione.

♦ **Caricamento.** Se nella fase di compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare **codice rilocabile**. In questo caso si ritarda l'associazione finale degli indirizzi alla fase del caricamento. Se l'indirizzo iniziale cambia, è sufficiente ricaricare il codice utente per incorporare il valore modificato.

♦ **Esecuzione.** Se durante l'esecuzione il processo può essere spostato da un segmento di memoria a un altro, si deve ritardare l'associazione degli indirizzi fino alla fase d'esecuzione. Per realizzare questo schema sono

necessarie specifiche caratteristiche dell'architettura; questo argomento è trattato nel Paragrafo 8.1.3. La maggior parte dei sistemi operativi d'uso generale impiega questo metodo.

Spazi di indirizzi logici e fisici a confronto

Un indirizzo generato dalla CPU di solito si indica come **indirizzo logico**, mentre un indirizzo visto dall'unità di memoria, cioè caricato nel **registro dell'indirizzo di memoria** (*memory address register*, MAR) di solito si indica come **indirizzo fisico**.

I metodi di associazione degli indirizzi nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Con i metodi di associazione nella fase d'esecuzione, invece, gli indirizzi logici non coincidono con gli indirizzi fisici. L'insieme di tutti gli indirizzi logici generati da un programma è lo **spazio degli indirizzi logici**; l'insieme degli indirizzi fisici corrispondenti a tali indirizzi logici è lo **spazio degli indirizzi fisici**. Quindi, con lo schema di associazione degli indirizzi nella fase d'esecuzione, lo spazio degli indirizzi logici differisce dallo spazio degli indirizzi fisici.

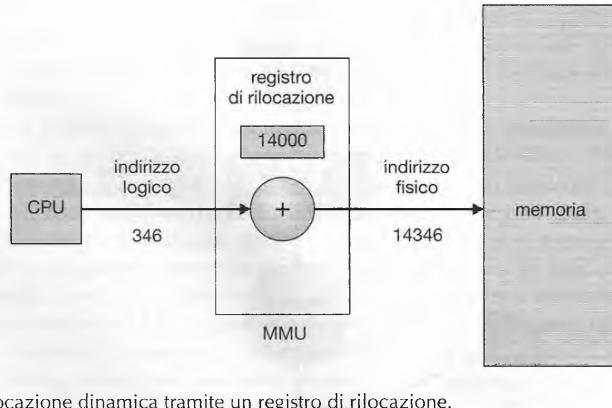


Figura 8.4 Rilocazione dinamica tramite un registro di rilocazione.

L'associazione nella fase d'esecuzione dagli indirizzi virtuali agli indirizzi fisici è svolta da un dispositivo detto **unità di gestione della memoria** (*memory-management unit*, MMU). Com'è illustrato nella Figura 8.4, il registro di base è ora denominato **registro di rilocazione**: quando un processo utente genera un indirizzo, prima dell'invio all'unità di memoria, si **SOMMA** a tale indirizzo il valore contenuto nel registro di rilocazione. Ad esempio, se il registro di rilocazione contiene il valore 14000, un tentativo da parte dell'utente di accedere alla locazione 0 è dinamicamente rilocalizzato alla locazione 14000; un accesso alla locazione 346 corrisponde alla locazione 14346.

Il programma utente non considera mai gli indirizzi fisici *reali*. Il programma crea un puntatore alla locazione 346, lo memorizza, lo modifica, lo confronta con altri indirizzi, tutto ciò semplicemente come un numero. Solo quando assume il ruolo di un indirizzo di memoria (magari in una load o una store indiretta), si riloca il numero sulla base del contenuto del registro di rilocazione. Il programma utente tratta indirizzi *logici*, l'architettura del sistema converte gli indirizzi logici in indirizzi fisici. In questo caso esistono due diversi tipi di indirizzi: gli indirizzi logici (nell'intervallo da 0 a *max*) e gli indirizzi fisici (nell'intervallo da $r + 0$ a $r + max$ per un valore di base r). L'utente genera solo indirizzi logici e *pensa* che il processo sia eseguito nelle posizioni da 0 a *max*. Il programma utente fornisce indirizzi logici che, prima d'essere usati, si devono far corrispondere a indirizzi fisici. Il concetto di *spazio d'indirizzi logici* associato a uno *spazio d'indirizzi fisici* separato è fondamentale per una corretta gestione della memoria.

Caricamento dinamico

Per migliorare l'utilizzo della memoria si può ricorrere al **caricamento dinamico** (*dynamic loading*), mediante il quale si carica una procedura solo quando viene richiamata; tutte le procedure si tengono in memoria secondaria in un formato di caricamento rilocalizzabile. Si carica il programma principale in memoria e quando, durante l'esecuzione, una procedura deve richiamarne un'altra, si controlla innanzitutto che sia stata caricata, altrimenti si richiama il caricatore di collegamento rilocalizzabile per caricare in memoria la procedura richiesta e aggiornare le tabelle degli indirizzi del programma in modo che registrino questo cambiamento. A questo punto il controllo passa alla procedura appena caricata. Il vantaggio dato dal caricamento dinamico consiste nel fatto che una procedura che non si adopera non viene caricata. Questo metodo è utile soprattutto quando servono grandi quantità di codice per gestire casi non frequenti, per esempio le procedure di gestione degli errori. In questi casi, anche se un programma può avere dimensioni totali elevate, la parte effettivamente usata, e quindi effettivamente caricata, può essere molto più piccola. Il caricamento dinamico non richiede un intervento particolare del sistema operativo. Spetta agli utenti progettare i programmi in modo da trarre vantaggio da un metodo di questo tipo. Il sistema operativo può tuttavia aiutare il programmatore fornendo librerie di procedure che realizzano il caricamento dinamico.

Collegamento dinamico e librerie condivise

La Figura 8.3 mostra anche le librerie collegate dinamicamente. Alcuni sistemi operativi consentono solo il collegamento statico, in cui le librerie di sistema del linguaggio sono trattate come qualsiasi altro modulo oggetto e combinate dal caricatore nell'immagine binaria del programma. Il concetto di collegamento dinamico è analogo a quello di caricamento dinamico. Invece di differire il caricamento di una procedura fino al momento dell'esecuzione, si differisce il collegamento. Questa caratteristica si usa soprattutto con le librerie di sistema, per esempio le librerie di procedure del linguaggio. Senza questo strumento tutti i programmi di un sistema dovrebbero disporre all'interno dell'immagine eseguibile di una copia della libreria di linguaggio (o almeno delle procedure cui il programma fa riferimento). Tutto ciò richiede spazio nei dischi e in memoria centrale. Con il collegamento dinamico, invece, per ogni riferimento a una procedura di libreria s'inserisce all'interno dell'immagine eseguibile una piccola porzione di codice di riferimento (*stub*), che indica come localizzare la giusta procedura di libreria residente in memoria o come caricare la libreria se la procedura non è già presente. Durante l'esecuzione, il codice di riferimento controlla se la procedura richiesta è già in memoria, altrimenti provvede a caricarla; in ogni caso tale codice sostituisce se stesso con l'indirizzo della procedura, che viene poi eseguita. In questo modo, quando si raggiunge nuovamente quel segmento del codice di riferimento, si esegue direttamente la procedura di libreria, senza costi aggiuntivi per il collegamento dinamico. Con questo metodo tutti i processi che usano una libreria del linguaggio si limitano a eseguire la stessa copia del codice della libreria. Questo sistema è noto anche con il nome di **librerie condivise**. A differenza del caricamento dinamico, il collegamento dinamico richiede generalmente l'assistenza del sistema operativo. Se i processi presenti in memoria sono protetti l'uno dall'altro, il sistema operativo è l'unica entità che può controllare se la procedura richiesta da un processo è nello spazio di memoria di un altro processo, o che può consentire l'accesso di più processi agli stessi indirizzi di memoria. Questo concetto è sviluppato nel contesto della **paginazione**.

Avvicendamento dei processi (swapping)

Per essere eseguito, un processo deve trovarsi in memoria centrale, ma si può trasferire temporaneamente in **memoria ausiliaria** (*backing store*) da cui si riporta in memoria centrale al momento di riprenderne l'esecuzione. Si consideri, per esempio, un ambiente di multiprogrammazione con un algoritmo circolare (*round-robin*) per lo scheduling della CPU. Tra scorso un quanto di tempo, il gestore di memoria scarica dalla memoria il processo appena terminato e carica un altro processo nello spazio di memoria appena liberato; questo procedimento è illustrato nella Figura 8.5 e si chiama **avvicendamento dei processi in memoria** - o, più brevemente,

avvicendamento o scambio (*swapping*). Nel frattempo lo scheduler della CPU assegna un quanto di tempo a un altro processo presente in memoria. Quando esaurisce il suo quanto di tempo, ciascun processo viene scambiato con un altro processo. In teoria il gestore della memoria può avvicendare i processi in modo sufficientemente rapido da far sì che alcuni siano presenti in memoria, pronti per essere eseguiti, quando lo scheduler della CPU voglia riassegnare la CPU stessa. Anche il quanto di tempo deve essere sufficientemente lungo da permettere che un processo, prima d'essere sostituito, esegua quantità ragionevole di calcolo.

Una variante di questo criterio d'avvicendamento dei processi s'impiega per gli algoritmi di scheduling basati sulle priorità. Se si presenta un processo con priorità

maggiori, il gestore della memoria può scaricare dalla memoria centrale il processo con priorità inferiore per fare spazio all'esecuzione del processo con priorità maggiore. Quando il processo con priorità maggiore termina, si può ricaricare in memoria quello con priorità minore e continuare la sua esecuzione (questo tipo d'avvicendamento è talvolta chiamato *roll out, roll in*). Normalmente, un processo che è stato scaricato dalla memoria si deve ricaricare nello spazio di memoria occupato in precedenza. Questa limitazione è dovuta al metodo di associazione degli indirizzi. Se l'associazione degli indirizzi logici agli indirizzi fisici della memoria si effettua nella fase di assemblaggio o di caricamento, il processo non può essere ricaricato in posizioni diverse.

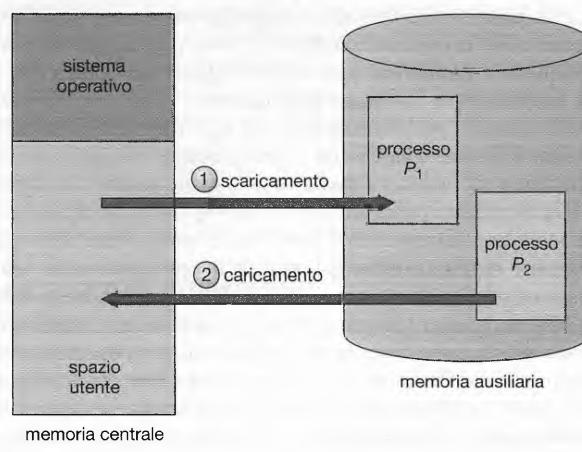


Figura 8.5 Avvicendamento di due processi con un disco come memoria ausiliaria.

Se l'associazione degli indirizzi logici agli indirizzi fisici della memoria si compie nella fase d'esecuzione, un processo può essere riversato in uno spazio di memoria diverso, poiché gli indirizzi fisici si calcolano nella fase d'esecuzione. L'avvicendamento dei processi richiede una memoria ausiliaria. Tale memoria ausiliaria deve essere abbastanza ampia da contenere le copie di tutte le immagini di memoria di tutti i processi utenti, e deve permettere un accesso diretto a dette immagini di memoria. Il sistema mantiene una coda dei processi pronti (*ready queue*) formata da tutti i processi pronti per l'esecuzione, le cui immagini di memoria si trovano in memoria ausiliaria o in memoria. Quando lo scheduler della **CPU** decide di eseguire un processo, richiama il dispatcher, che controlla se il primo processo della coda si trova in memoria. Se non si trova in memoria, e in questa non c'è spazio libero, il dispatcher scarica un processo dalla memoria e vi carica il processo richiesto dallo scheduler della **CPU**, quindi ricarica normalmente i registri e trasferisce il controllo al processo selezionato.

Allocazione contigua della memoria

La memoria centrale deve contenere sia il sistema operativo sia i vari processi utenti; perciò è necessario assegnare le diverse parti della memoria centrale nel modo più efficiente. La memoria centrale di solito si divide in due partizioni, una per il sistema operativo residente e una per i processi utenti. Il sistema operativo si può collocare sia in memoria bassa sia in memoria alta. Il fattore che incide in modo decisivo su tale scelta è generalmente la posizione del vettore delle interruzioni. Poiché si trova spesso in memoria bassa, i programmati collocano di solito anche il sistema operativo in memoria bassa. Per questo motivo prendiamo in considerazione solo la situazione in cui il sistema operativo risiede in quest'area di memoria. Generalmente si vuole che più processi utenti risiedano contemporaneamente in memoria centrale. Perciò è necessario considerare come assegnare la memoria disponibile ai processi presenti nella coda d'ingresso che attendono di essere caricati in memoria. Con l'**alocazione contigua della memoria**, ciascun processo è contenuto in una singola sezione contigua di memoria.

Rilocazione e protezione della memoria

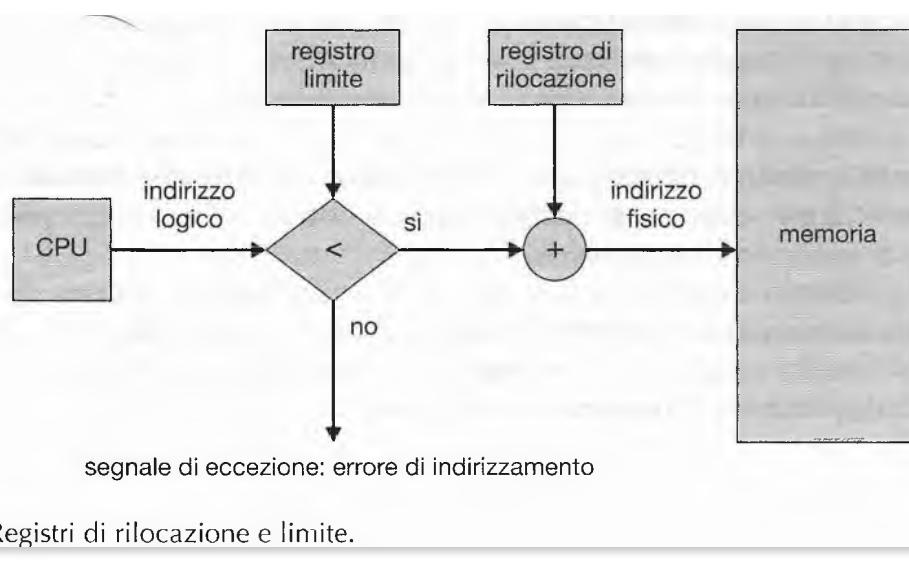


Figura 8.6 Registri di rilocazione e limite.

Il registro di rilocazione contiene il valore dell'indirizzo fisico minore; il registro limite contiene l'intervallo di indirizzi logici, per esempio, *rilocazione* = 100.040 e *limite* = 74.600. Con i registri di rilocazione e limite, ogni indirizzo logico deve essere minore del contenuto del registro limite; la **MMU** fa corrispondere *dinamicamente* l'indirizzo fisico all'indirizzo logico sommando a quest'ultimo il valore contenuto nel registro di rilocazione (Figura 8.6).

Quando lo scheduler della **CPU** seleziona un processo per l'esecuzione, il dispatcher, durante l'esecuzione del cambio di contesto, carica il registro di rilocazione e il registro limite con i valori corretti. Poiché si confronta ogni indirizzo generato dalla **CPU** con i valori contenuti in questi registri, si possono proteggere il sistema operativo, i programmi e i dati di altri utenti da tentativi di modifiche da parte del processo in esecuzione. Lo schema con registro di rilocazione consente al sistema operativo di cambiare dinamicamente le proprie dimensioni. Tale flessibilità è utile in molte situazioni; il sistema operativo, per esempio, contiene codice e spazio di memoria per i driver dei dispositivi; se uno di questi, o un altro servizio del sistema operativo, non è comunemente usato, è inutile tenerne in memoria codice e dati, poiché lo spazio occupato si potrebbe usare per altri scopi.

Allocazione della memoria

Quando entrano nel sistema, i processi vengono inseriti in una coda d'ingresso. Per determinare a quali processi si debba assegnare la memoria, il sistema operativo tiene conto dei requisiti di memoria di ciascun processo e della quantità di spazio di memoria disponibile. Quando a un processo si assegna dello spazio, il processo stesso viene

caricato in memoria e può quindi competere per il controllo della CPU. Al termine, rilascia la memoria che gli era stata assegnata, e il sistema operativo può impiegarla per un altro processo presente nella coda d'ingresso.

In ogni dato istante è sempre disponibile una lista delle dimensioni dei blocchi liberi e della coda d'ingresso. Il sistema operativo può ordinare la coda d'ingresso secondo un algoritmo di scheduling. La memoria si assegna ai processi della coda finché si possono soddisfare i requisiti di memoria del processo successivo, cioè finché esiste un blocco di memoria (o buco) disponibile, sufficientemente grande da accogliere quel processo. Il sistema operativo può quindi attendere che si renda disponibile un blocco sufficientemente grande, oppure può scorrere la coda d'ingresso per verificare se sia possibile soddisfare le richieste di memoria più limitate di qualche altro processo. In generale, è sempre presente un *insieme* di buchi di diverse dimensioni sparsi per la memoria. Quando si presenta un processo che necessita di memoria, il sistema cerca nel gruppo un buco di dimensioni sufficienti per contenerlo. Se è troppo grande, il buco viene diviso in due parti: si assegna una parte al processo in arrivo e si riporta l'altra nell'insieme dei buchi. Quando termina, un processo rilascia il blocco di memoria, che si reinserisce nel l'insieme dei buchi; se si trova accanto ad altri buchi, si uniscono tutti i buchi adiacenti per formarne uno più grande. A questo punto il sistema deve controllare se vi siano processi nel l'attesa di spazio di memoria, e se la memoria appena liberata e ricombinata possa soddisfare le richieste di qualcuno fra tali processi. Questa procedura è una particolare istanza del più generale problema di **allocazione dinamica della memoria**, che consiste nel soddisfare una richiesta di dimensione n data una lista di buchi liberi. Le soluzioni sono numerose. I criteri più usati per scegliere un buco libero tra quelli disponibili nell'insieme sono i seguenti.

- **First-fit.** Si assegna il *primo* buco abbastanza grande. La ricerca può cominciare sia dal l'inizio dell'insieme di buchi sia dal punto in cui era terminata la ricerca precedente. Si può fermare la ricerca non appena s'individua un buco libero di dimensioni sufficientemente grandi.
- ♦ **Best-fit.** Si assegna il *più piccolo* buco in grado di contenere il processo. Si deve compiere la ricerca in tutta la lista, sempre che questa non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più piccole.
- ♦ **Worst-fit.** Si assegna il buco *più grande*. Anche in questo caso si deve esaminare tutta la lista, sempre che non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più grandi, che possono essere più utili delle parti più piccole ottenute col criterio precedente.

Frammentazione

Entrambi i criteri first-fit e best-fit di allocazione della memoria soffrono di **frammentazione esterna**: quando si caricano e si rimuovono i processi dalla memoria, si frammenta lo spazio libero della memoria in tante piccole parti. Si ha la frammentazione esterna se lo spazio di memoria totale è sufficiente per soddisfare una richiesta, ma non è contiguo; la memoria è frammentata in tanti piccoli buchi. Questo problema di frammentazione può essere molto grave; nel caso peggiore può verificarsi un blocco di memoria libera, sprecata, tra ogni coppia di processi. Se tutti questi piccoli pezzi di memoria costituissero in un unico blocco libero di grandi dimensioni, si potrebbero eseguire molti più processi. Sia che si adotti l'uno o l'altro, l'impiego di un determinato criterio può influire sulla quantità di frammentazione: in alcuni sistemi dà migliori risultati la scelta del primo buco abbastanza grande, in altri dà migliori risultati la scelta del più piccolo tra i buchi abbastanza grandi; inoltre è necessario sapere qual è l'estremità assegnata di un blocco libero (se la parte inutilizzata è quella in alto o quella in basso). A prescindere dal tipo di algoritmo usato, la frammentazione esterna è un problema. La sua gravità dipende dalla quantità totale di memoria e dalla dimensione media dei processi. La frammentazione può essere sia interna sia esterna. Si consideri il metodo d'allocazione con più partizioni con un buco di 18.464 byte. Supponendo che il processo successivo richieda 18.462 byte, assegnando esattamente il blocco richiesto rimane un buco di 2 byte. Il carico necessario per tener traccia di questo buco è sostanzialmente più grande del buco stesso. Il metodo generale prevede di suddividere la memoria fisica in blocchi di dimensione fissa, che costituiscono le unità d'allocazione. Con questo metodo la memoria assegnata può essere leggermente maggiore della memoria richiesta.

La **frammentazione interna** consiste nella differenza tra questi due numeri; la memoria è interna a una partizione, ma non è in uso. Una soluzione al problema della frammentazione esterna è data dalla compattazione. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco. La compattazione tuttavia non è sempre possibile: non si può realizzare se la rilocazione è statica ed è fatta nella fase di assemblaggio o di caricamento; è possibile solo se la rilocazione è dinamica e si compie nella fase d'esecuzione. Se gli indirizzi sono rilocati dinamicamente, la rilocazione richiede solo lo spostamento del programma e dei dati, e quindi la modifica del registro di rilocazione in modo che rifletta il nuovo indirizzo di base. Quando è possibile

eseguire la compattazione, è necessario determinarne il costo. Il più semplice algoritmo di compattazione consiste nello spostare tutti i processi verso un'estremità della memoria, mentre tutti i buchi vengono spostati nell'altra direzione for mando un grosso buco di memoria. Questo metodo può essere assai oneroso. Un'altra possibile soluzione del problema della frammentazione esterna è data dal con sentire la non contiguità dello spazio degli indirizzi logici di un processo, permettendo così di assegnare la memoria fisica ai processi dovunque essa sia disponibile. Due tecniche complementari conseguono questo risultato: la paginazione (Paragrafo 8.4) e la segmentazione (Paragrafo 8.6). Queste tecniche si possono anche combinare (Paragrafo 8.7).

Paginazione

La paginazione è un metodo di gestione della memoria che permette che lo spazio degli indirizzi fisici di un processo non sia contiguo. Elimina il gravoso problema della sistemazione di blocchi di memoria di diverse dimensioni in memoria ausiliaria, una questione che riguarda la maggior parte dei metodi di gestione della memoria analizzati. Il problema insorge perché, quando alcuni frammenti di codice o dati residenti in memoria centrale devono essere scaricati, si deve trovare lo spazio necessario in memoria ausiliaria. I problemi di frammentazione relativi alla memoria centrale valgono anche per la memoria ausiliaria, con la differenza che in questo caso l'accesso è molto più lento, quindi è impossibile eseguire la compattazione.

Metodo di base

Il metodo di base per implementare la paginazione consiste nel suddividere la memoria fisica in blocchi di dimensione costante, detti anche frame o pagine fisiche, e nel suddividere la memoria logica in blocchi di pari dimensione, detti pagine. Quando si deve eseguire un processo, si caricano le sue pagine nei frame disponibili, prendendole dalla memoria ausiliaria, divisa in blocchi di dimensione fissa, uguale a quella dei frame della memoria.

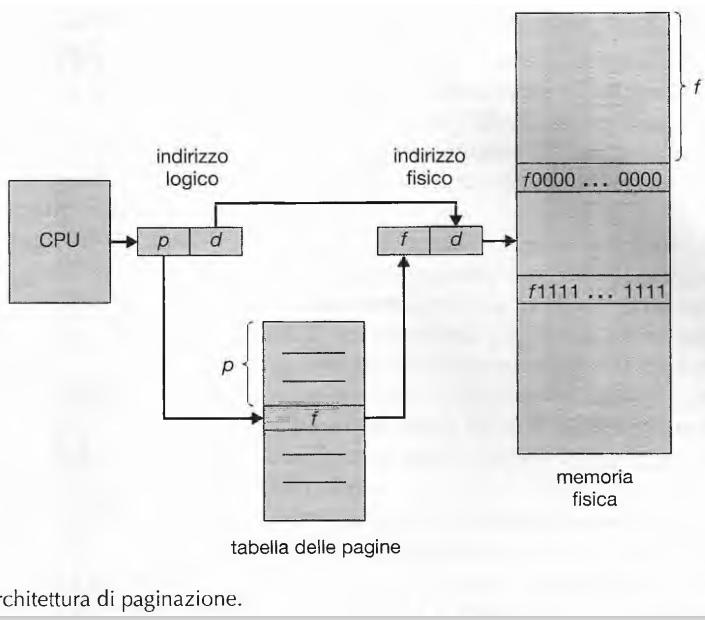


Figura 8.7 Architettura di paginazione.

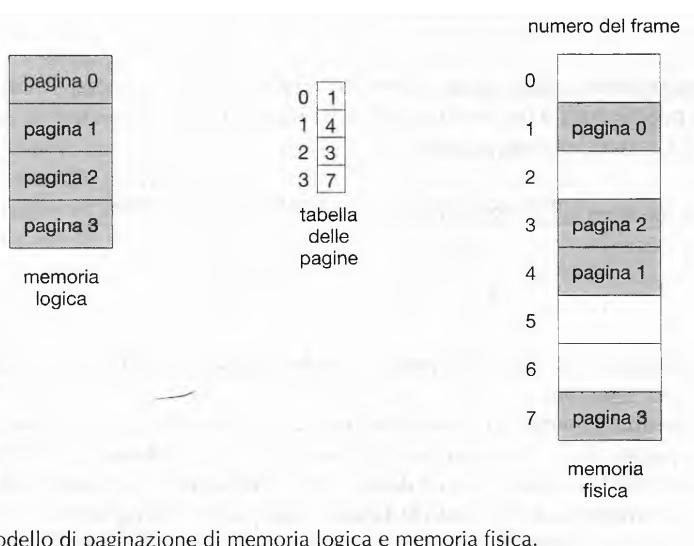


Figura 8.8 Modello di paginazione di memoria logica e memoria fisica.

Ogni indirizzo generato dalla CPU è diviso in due parti: un numero di pagina (p), e uno scostamento (offset) di pagina (d). Il numero di pagina serve come indice per la tabella delle pagine, contenente l'indirizzo di base in memoria fisica di ogni pagina. Questo indirizzo di base si combina con lo scostamento di pagina per definire l'indirizzo della memoria fisica, che s'invia all'unità di memoria. La dimensione di una pagina, così come quella di un frame, è definita dall'architettura del calcolatore ed è, in genere, una potenza di 2 compresa tra 512 byte e 16 MB.

La scelta di una potenza di 2 come dimensione della pagina facilita notevolmente la traduzione di un indirizzo logico nei corrispondenti numero di pagina e scostamento di pagina. Se la dimensione dello spazio degli indirizzi logici è 2^m e la dimensione di una pagina è di 2^n unità di indirizzamento (byte o parole), allora gli $m - n$ bit più significativi di un indirizzo logico indicano il numero di pagina, e gli n bit meno significativi indicano lo scostamento di pagina.

L'indirizzo logico ha quindi la forma seguente: dove p è un indice della tabella delle pagine e d è lo scostamento all'interno della pagina indicata da p .

Esempio:

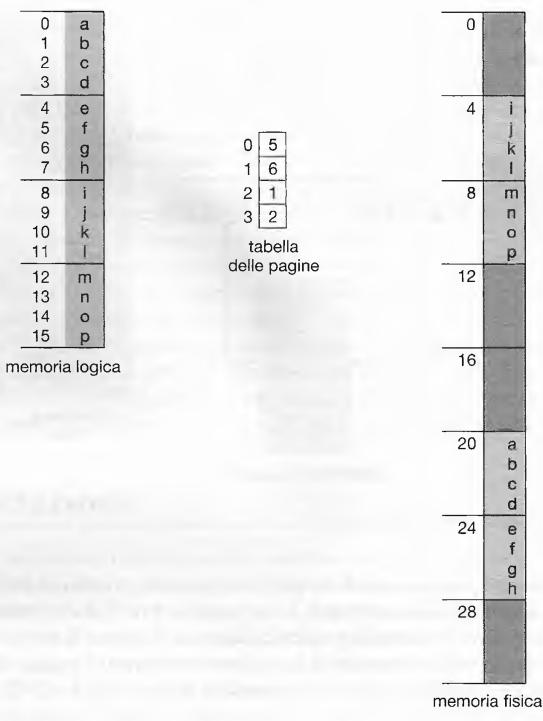


Figura 8.9 Esempio di paginazione per una memoria di 32 byte con pagine di 4 byte.

Si consideri la memoria illustrata nella Figura 8.9; con pagine di 4 byte e una memoria fisica di 32 byte (8 pagine), si mostra come si faccia corrispondere la memoria vista dall’utente alla memoria fisica. L’indirizzo logico 0 è la pagina 0 con scostamento 0. Secondo la tabella delle pagine, la pagina 0 si trova nel frame 5. Quindi all’indirizzo logico 0 corrisponde l’indirizzo fisico 20 ($= (5 \times 4) + 0$).

All’indirizzo logico 3 (pagina 0, scostamento 3) corrisponde l’indirizzo fisico 23 ($= (5 \times 4) + 3$). Per quel che riguarda l’indirizzo logico 4 (pagina 1, scostamento 0), secondo la tabella delle pagine, alla pagina 1 corrisponde il frame 6, quindi, all’indirizzo logico 4 corrisponde l’indirizzo fisico 24 ($= (6 \times 4) + 0$). All’indirizzo logico 13 corrisponde l’indirizzo fisico 9. Il lettore può aver notato che la paginazione non è altro che una forma di rilocazione dinamica: a ogni indirizzo logico l’architettura di paginazione fa corrispondere un indirizzo fisico. L’uso della tabella delle pagine è simile all’uso di una tabella di registri base (o di rilocazione), uno per ciascun frame. Con la paginazione si può evitare la frammentazione esterna: *qualsiasi* frame libero si può

assegnare a un processo che ne abbia bisogno; tuttavia si può avere la frammentazione interna. I frame si assegnano come unità. Poiché in generale lo spazio di memoria richiesto da un processo non è un multiplo delle dimensioni delle pagine, l’ultimo frame assegnato può non essere completamente pieno. Se, per esempio, le pagine sono di 2048 byte, un processo di 72.766 byte necessita di 35 pagine più 1086 byte. Si assegnano 36 frame, quindi si ha una frammentazione interna di $2048 - 1086 = 962$ byte. Il caso peggiore si ha con un processo che necessita di n pagine più un byte: si assegnano $n + 1$ frame, quindi si ha una frammentazione interna di quasi un intero frame. Se la dimensione del processo è indipendente dalla dimensione della pagina, ci si deve aspettare una frammentazione interna media di mezza pagina per processo. Questa considerazione suggerisce che conviene usare pagine di piccole dimensioni; tuttavia, a ogni elemento della tabella delle pagine è associato un carico che si può ridurre aumentando le dimensioni delle pagine.

Quando si deve eseguire un processo, si esamina la sua dimensione espressa in pagine. Poiché ogni pagina del processo necessita di un frame, se il processo richiede n pagine, devono essere disponibili almeno n frame che, se ci sono, si assegnano al processo stesso. Si carica la prima pagina del processo in uno dei frame assegnati e s’inscrive il numero del frame nella tabella delle pagine relativa al processo in questione.

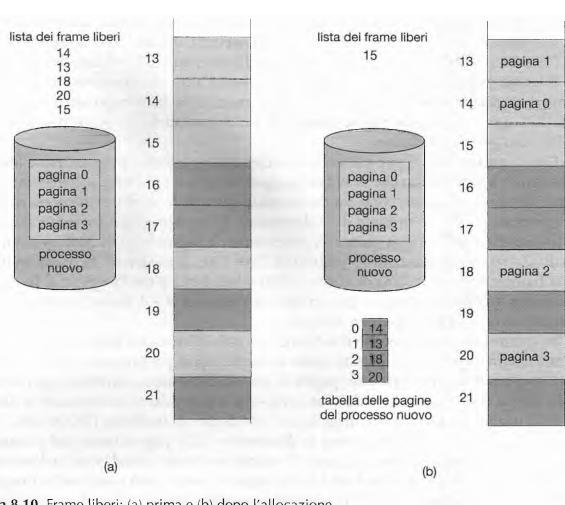


Figura 8.10 Frame liberi; (a) prima e (b) dopo l’allocazione.

La pagina successiva si carica in un altro frame e, anche in questo caso, s’inscrive il numero del frame nella tabella delle pagine, e così via (Figura 8.10).

Un aspetto importante della paginazione è la netta distinzione tra la memoria vista dal l’utente e l’effettiva memoria fisica: il programma utente *vede* la memoria come un unico spazio contiguo, contenente solo il programma stesso; in realtà, il programma utente è sparso in una memoria fisica contenente anche altri programmi. La differenza tra la memoria vista dall’utente e la memoria fisica è colmata dall’architettura di traduzione degli indirizzi, che fa corrispondere gli indirizzi fisici agli indirizzi logici generati dai processi utenti. Queste trasformazioni non sono visibili agli utenti e sono controllate dal sistema operativo.

Architettura di paginazione

Ogni sistema operativo segue metodi propri per memorizzare le tabelle delle pagine. La maggior parte dei sistemi impiega una tabella delle pagine per ciascun processo. Il PCB contiene, insieme col valore di altri registri, come il registro delle istruzioni, un puntatore alla tabella delle pagine. Per avviare un processo, il dispatcher ricarica i registri utente e imposta i corretti valori della tabella delle pagine fisiche, usando la tabella delle pagine presente in memoria e relativa al processo. L'uso di registri per la tabella delle pagine è efficiente se la tabella stessa è ragionevolmente piccola, nell'ordine, per esempio, di 256 elementi. La maggior parte dei calcolatori contemporanei usa comunque tabelle molto grandi, per esempio di un milione di elementi, quindi non si possono impiegare i registri veloci per realizzare la tabella delle pagine; quest'ultima si mantiene in memoria principale e un **registro di base della tabella delle pagine** (*page-table base register*, PTBR) punta alla tabella stessa. Il cambio delle tabelle delle pagine richiede soltanto di modificare questo registro, riducendo considerevolmente il tempo dei cambi di contesto. Questo metodo presenta un problema connesso al tempo necessario di accesso a una locazione della memoria utente. Per accedere alla locazione z , occorre far riferimento alla tabella delle pagine usando il valore contenuto nel PTBR aumentato del numero di pagina relativo a i , perciò si deve accedere alla memoria. Si ottiene il numero del frame che, associato allo scostamento di pagina, produce l'indirizzo cercato; a questo punto è possibile accedere alla posizione di memoria desiderata. Con questo metodo, per accedere a un byte occorrono *due* accessi alla memoria (uno per l'elemento della tabella delle pagine e uno per il byte stesso), quindi l'accesso alla memoria è rallentato di un fattore 2.

Nella maggior parte dei casi un tale ritardo è intollerabile; sarebbe più conveniente ricorrere all'avvicendamento dei processi!

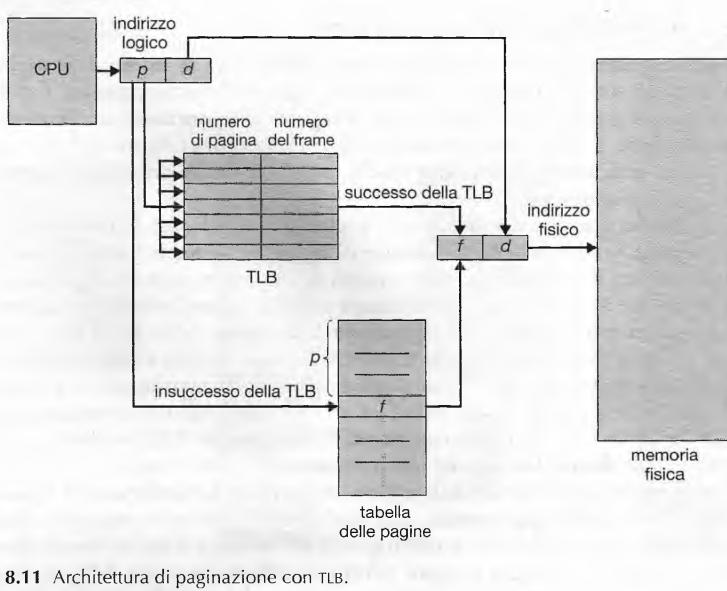


Figura 8.11 Architettura di paginazione con TLB.

La soluzione tipica a questo problema consiste nell'impiego di una speciale, piccola cache di ricerca veloce, detta **TLB** (*translation look-aside buffer*).

La TLB è una memoria associativa ad alta velocità in cui ogni suo elemento consiste di due parti: una chiave, o un indicatore (*tag*) e un valore. Quando si presenta un elemento, la memoria associativa lo confronta contemporaneamente con tutte le chiavi; se trova una corrispondenza, riporta il valore correlato. La ricerca è molto rapida, ma le memorie associative sono molto costose. Se la ricerca nella TLB richiede 20 nanosecondi e sono necessari 100 nano secondi per accedere alla memoria, allora, supponendo che il numero di pagina si trovi nella TLB, un accesso alla memoria richiede 120 nanosecondi. Se, invece, il numero non è contenuto nella TLB (20 nanosecondi), occorre accedere alla memoria per arrivare alla tabella delle pagine e al numero del frame (100 nanosecondi), quindi accedere al byte desiderato in memoria (100 nanosecondi); in totale sono necessari 220 nanosecondi.

Per calcolare il tempo effettivo d'accesso alla memoria occorre tener conto della probabilità dei due casi:

tempo effettivo d'accesso $\rightarrow 0,80 \times 120 + 0,20 \times 220 = 140$ nanosecondi

In questo esempio si verifica un rallentamento del 40 per cento nel tempo d'accesso alla memoria (da 100 a 140 nanosecondi).

Per un tasso di successi del 98 per cento si ottiene il seguente risultato:

tempo effettivo d'accesso $\rightarrow 0,98 \times 120 + 0,02 \times 220 = 122$ nanosecondi

Aumentando il tasso di successi, il rallentamento del tempo d'accesso alla memoria scende al 22 per cento.

Protezione

In un ambiente paginato, la protezione della memoria è assicurata dai bit di protezione associati a ogni frame; normalmente tali bit si trovano nella tabella delle pagine. Un bit può determinare se una pagina si può leggere e scrivere oppure soltanto leggere. Di solito si associa a ciascun elemento della tabella delle pagine un ulteriore bit, detto bit di validità. Tale bit, impostato a *valido*, indica che la pagina corrispondente è nello spazio d'indirizzi logici

del processo, quindi è una pagina valida; impostato a *non valido*, indica che la pagina non è nello spazio d'indirizzi logici del processo.

Pagine condivise

Un altro vantaggio della paginazione consiste nella possibilità di *condividere* codice comune. Questa considerazione è importante soprattutto in un ambiente a partizione del tempo. Il codice rientrante, detto anche codice puro, è un codice non automodificante: non cambia durante l'esecuzione. Quindi, due o più processi possono eseguire lo stesso codice nello stesso momento. Ciascun processo dispone di una propria copia dei registri e di una memoria dove conserva i dati necessari alla propria esecuzione. I dati per due differenti processi variano, ovviamente, per ciascun processo. In memoria fisica è presente una sola copia dell'elaboratore di testi: la tabella delle pagine di ogni utente fa corrispondere gli stessi frame contenenti l'elaboratore di testi, mentre le pagine dei dati si fanno corrispondere a frame diversi.

Struttura della tabella delle pagine:

- Paginazione gerarchica

La maggior parte dei moderni calcolatori dispone di uno spazio d'indirizzi logici molto grande (da 2³² a 2⁶⁴ elementi). In un ambiente di questo tipo la stessa tabella delle pagine finirebbe per diventare eccessivamente grande. Si consideri, per esempio, un sistema con uno spazio d'indirizzi logici a 32 bit. Se la dimensione di ciascuna pagina è di 4 KB (2¹²), la tabella delle pagine potrebbe arrivare a contenere fino a 1 milione di elementi (2³²/2¹²). Se ogni elemento consiste di 4 byte, ciascun processo potrebbe richiedere fino a 4 MB di spazio fisico d'indirizzi solo per la tabella delle pagine. Chiaramente, sarebbe meglio evitare di collocare la tabella delle pagine in modo contiguo in memoria centrale. Una semplice soluzione a questo problema consiste nel suddividere la tabella delle pagine in parti più piccole; questo risultato si può ottenere in molti modi.

Un metodo consiste nell'adottare un algoritmo di paginazione a due livelli, in cui la tabella stessa è paginata (Figura 8.14). Si consideri il precedente esempio di macchina a 32 bit con dimensione delle pagine di 4 KB. Si suddivide ciascun indirizzo logico in un numero di pagina di 20 bit e in uno scostamento di pagina di 12 bit. Paginando la tabella delle pagine, anche il numero di pagina è a sua volta suddiviso in un numero di pagina di 10 bit e uno scostamento di pagina di 10 bit. Quindi, l'indirizzo logico è:

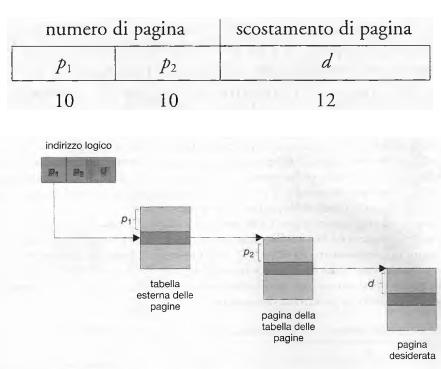


Figura 8.15 Traduzione degli indirizzi per un'architettura a 32 bit con paginazione a due livelli.

dove p_1 è un indice della tabella delle pagine di primo livello, o tabella esterna delle pagine, e p_2 è lo scostamento all'interno della pagina indicata dalla tabella esterna delle pagine.

Il metodo di traduzione degli indirizzi seguito da questa architettura è mostrato nella Figura 8.15. Poiché la traduzione degli indirizzi si svolge dalla tabella esterna delle pagine verso l'interno, questo metodo è anche noto come tabella delle pagine ad associazione diretta (*forward-mapped page table*).

- Tabella delle pagine di tipo hash

Un metodo di gestione molto comune degli spazi d'indirizzi relativi ad architetture oltre i 32 bit consiste nell'impiego di una **tabella delle pagine di tipo hash**.

In questo tipo di tabella delle pagine, la funzione hash è il numero della pagina virtuale. Ciascun elemento è composto da tre campi: (1) il numero della pagina virtuale; (2) l'indirizzo del frame (pagina fisica) corrispondente alla pagina virtuale; (3) un puntatore al successivo elemento della lista.

L'algoritmo opera come segue: si applica la funzione hash al numero della pagina virtuale contenuto nell'indirizzo virtuale, identificando un elemento della tabella. Si confronta il numero di pagina virtuale con il campo (1) del primo elemento della lista concatenata corrispondente. Se i valori coincidono, si usa l'indirizzo del relativo frame (campo 2) per generare l'indirizzo fisico desiderato. Altrimenti, l'algoritmo esamina allo stesso modo gli elementi successivi della lista concatenata (Figura 8.16). Le tabelle delle pagine di tipo hash sono particolarmente utili per

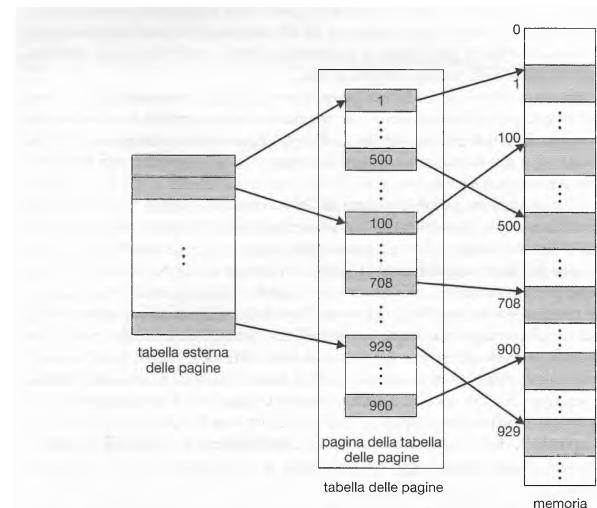


Figura 8.14 Schema di una tabella delle pagine a due livelli.

gli spazi d'indirizzi sparsi, in cui i riferimenti alla memoria non sono contigui ma distribuiti per tutto lo spazio d'indirizzi. Per questo schema è stata proposta una variante, adatta a spazi di indirizzamento a 64 bit. Si tratta della tabella delle pagine a gruppi (*clustered page table*), simile alla tabella hash;

Tabella delle pagine invertita

Generalmente, si associa una tabella delle pagine a ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando, oppure un elemento per ogni indirizzo virtuale a prescindere dalla validità di quest'ultimo. Questa è una rappresentazione naturale della tabella, poiché i processi fanno riferimento alle pagine tramite gli indirizzi virtuali delle pagine stesse, che il sistema operativo deve poi tradurre in indirizzi di memoria fisica. Poiché la tabella è ordinata per indirizzi virtuali, il sistema operativo può calcolare in che punto della tabella si trova l'elemento dell'indirizzo fisico associato, e usare direttamente tale valore.

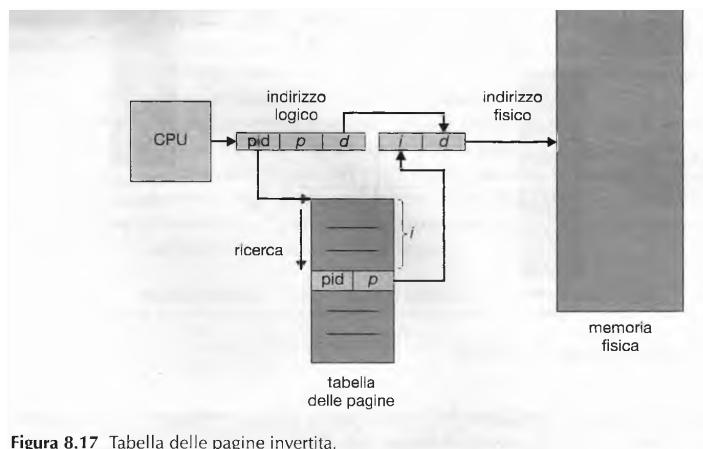


Figura 8.17 Tabella delle pagine invertita.

Per risolvere questo problema si può fare uso della tabella delle pagine invertita. Una tabella delle pagine invertita ha un elemento per ogni pagina reale (o frame). Ciascun elemento è quindi costituito dell'indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che possiede tale pagina. Quindi, nel sistema esiste una sola tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica. Nella Figura 8.17 sono mostrate le operazioni di una tabella delle pagine invertita; si confronti questa figura con la Figura 8.7, che illustra il modo di operare per una tabella delle pagine ordinaria.

Segmentazione

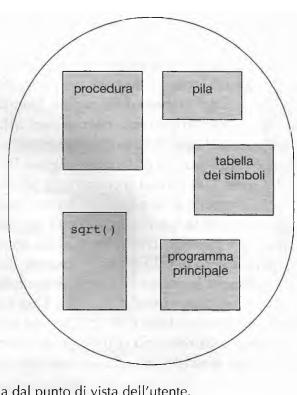


Figura 8.18 Un programma dal punto di vista dell'utente.

Un aspetto importante della gestione della memoria, inevitabile alla presenza della paginazione, è quello della separazione tra la visione della memoria dell'utente e l'effettiva memoria fisica. La **segmentazione** è uno schema di gestione della memoria che consente di gestire questa rappresentazione della memoria dal punto di vista dell'utente. Uno spazio d'indirizzi logici è una raccolta di segmenti, ciascuno dei quali ha un nome e una lunghezza. Gli indirizzi specificano sia il nome sia lo scostamento all'interno del segmento, quindi l'utente fornisce ogni indirizzo come una coppia ordinata di valori: un nome di segmento e uno scostamento. Questo schema contrasta con la paginazione, in cui l'utente fornisce un indirizzo singolo, che l'architettura di paginazione suddivide in un numero di pagine e uno scostamento, non visibili dal programmatore.

Per semplicità i segmenti sono numerati, e ogni riferimento si compie per mezzo di un numero anziché di un nome; quindi un indirizzo logico è una *coppia :<numero di segmento, scostamento>*. Sebbene l'utente possa far riferimento agli oggetti del programma per mezzo di un indirizzo bidimensionale, la memoria fisica è in ogni caso una sequenza di byte unidimensionale. Per questo motivo occorre tradurre gli indirizzi bidimensionali definiti dall'utente negli indirizzi fisici unidimensionali. Questa operazione si compie tramite una tabella dei segmenti; ogni suo elemento è una coppia ordinata: la *base del segmento* e il *limite del segmento*.

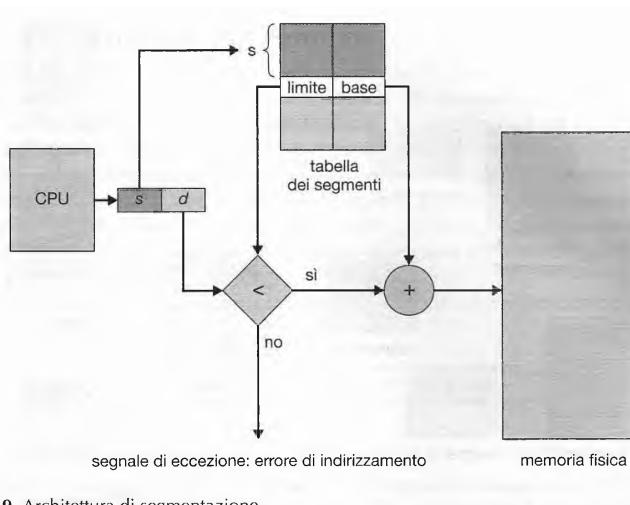


Figura 8.19 Architettura di segmentazione.

La base del segmento contiene l'indirizzo fisico iniziale della memoria dove il segmento risiede, mentre il limite del segmento contiene la lunghezza del segmento.

Sommario 8°CAPITOLO

Gli algoritmi di gestione della memoria per sistemi operativi multiprogrammati variano da un metodo semplice, per sistema con singolo utente, fino alla segmentazione paginata. Il fattore più rilevante che incide maggiormente sulla scelta del metodo da seguire in un sistema particolare è costituito dall'architettura disponibile. È necessario controllare la validità di ogni indirizzo di memoria generato dalla CPU; inoltre tali indirizzi, se sono corretti, devono essere tradotti in un indirizzo fisico. Tali controlli per essere efficienti devono essere effettuati dall'architettura del sistema, che secondo le sue caratteristiche pone vincoli al sistema operativo.

Gli algoritmi di gestione della memoria analizzati (allocazione contigua, paginazione, segmentazione e combinazione di paginazione e segmentazione) differiscono per molti aspetti. Per confrontare i diversi metodi di gestione della memoria si possono considerare i seguenti elementi.

- ♦ **Architettura.** Un semplice registro di base oppure una coppia di registri di base e limiti te è sufficiente per i metodi con partizione singola e con più partizioni, mentre la paginazione e la segmentazione necessitano di tabelle di traduzione per definire la corrispondenza degli indirizzi.
- ♦ **Prestazioni.** Aumentando la complessità dell'algoritmo, aumenta anche il tempo necessario per tradurre un indirizzo logico in un indirizzo fisico. Nei sistemi più semplici è sufficiente fare un confronto o sommare un valore all'indirizzo logico; si tratta di operazioni rapide. Paginazione e segmentazione possono essere altrettanto rapide se per realizzare la tabella s'impiegano registri veloci. Se però la tabella si trova in memoria, gli accessi alla memoria utente possono essere assai più lenti. Una TLB può limitare il calo delle prestazioni a un livello accettabile.
- ♦ **Frammentazione.** Un sistema multiprogrammato esegue le elaborazioni generalmente in modo più efficiente se ha un più elevato livello di multiprogrammazione. Per un dato gruppo di processi, il livello di multiprogrammazione si può aumentare solo compattando più processi in memoria. Per eseguire questo compito occorre ridurre lo spreco di memoria o la frammentazione. Sistemi con unità di allocazione di dimensione fissa, come lo schema con partizione singola e la paginazione, soffrono di frammentazione interna. Sistemi con unità di allocazione di dimensione variabile, come lo schema con più partizioni e la segmentazione, soffrono di frammentazione esterna.
- ♦ **Rilocazione.** Una soluzione al problema della frammentazione esterna è data dalla compattazione, che implica lo spostamento di un programma in memoria, senza che il programma stesso *si accorga* del cambiamento. Ciò richiede che gli indirizzi logici siano rilocati dinamicamente al momento dell'esecuzione. Se gli indirizzi si rilocano solo al momento del caricamento, non è possibile compattare la memoria.
- ♦ **Avvicendamento dei processi.** L'avvicendamento dei processi (*swapping*) si può incorporare in ogni algoritmo. I processi si copiano dalla memoria centrale alla memoria ausiliaria, e successivamente si ricopiano in memoria centrale a intervalli fissati dal sistema operativo, e generalmente stabiliti dai criteri di scheduling della CPU. Questo schema permette di inserire contemporaneamente in memoria più processi da eseguire.
- ♦ **Condivisione.** Un altro mezzo per aumentare il livello di multiprogrammazione è quello della condivisione del codice e dei dati tra diversi utenti. Poiché deve fornire piccoli pacchetti d'informazioni (pagine o segmenti) condivisibili, generalmente richiede l'uso della paginazione o della segmentazione. La condivisione permette di eseguire molti processi con una quantità di memoria limitata, ma i programmi e i dati condivisi si devono progettare con estrema cura.
- ♦ **Protezione.** Con la paginazione o la segmentazione, diverse sezioni di un programma utente si possono dichiarare di sola esecuzione, di sola lettura oppure di lettura e scrittura. Questa limitazione è necessaria per il codice e i dati condivisi, ed è utile, in genere, nei casi in cui sono richiesti semplici controlli nella fase d'esecuzione per l'individuazione degli errori di programmazione.

Capitolo 9

Memoria virtuale

La memoria virtuale è una tecnica che permette di eseguire processi che possono anche non essere completamente contenuti in memoria. Il vantaggio principale offerto da questa tecnica è quello di permettere che i programmi siano

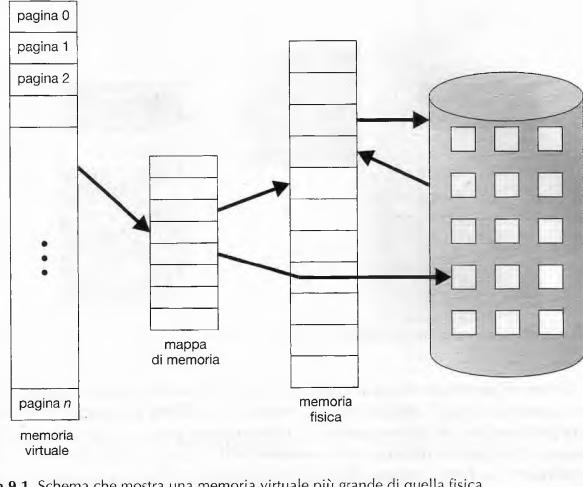


Figura 9.1 Schema che mostra una memoria virtuale più grande di quella fisica.

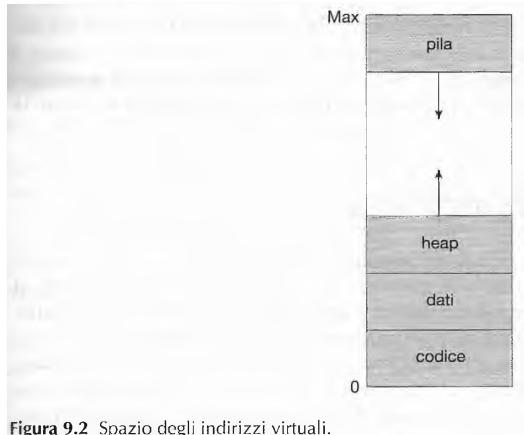


Figura 9.2 Spazio degli indirizzi virtuali.

più grandi della memoria fisica; inoltre la memoria virtuale astrae la memoria centrale in un vettore di memorizzazione molto grande e uniforme, separando la memoria logica, com'è vista dall'utente, da quella fisica. Questa tecnica libera i programmatori da quel che riguarda i limiti della memoria.

La memoria virtuale permette inoltre ai processi di condividere facilmente file e spazi d'indirizzi, e fornisce un meccanismo efficiente per la creazione dei processi. La **memoria virtuale** si fonda sulla separazione della memoria logica percepita dal l'utente dalla memoria fisica. Questa separazione permette di offrire ai programmatori una memoria virtuale molto ampia, anche se la memoria fisica disponibile è più piccola, com'è illustrato nella Figura 9.1.

L'espressione **spazio degli indirizzi virtuali** si riferisce alla collocazione dei processi in memoria dal punto di vista logico (o virtuale). Da tale punto di vista, un processo inizia in corrispondenza di un certo indirizzo logico - per esempio, l'indirizzo 0 - e si estende alla memoria contigua, come evidenziato dalla Figura 9.2.

Si noti come, nella Figura 9.2, allo heap sia lasciato sufficiente spazio per crescere verso l'alto nello spazio di memoria, poiché esso ospita la memoria allocata dinamicamente. In modo analogo, consentiamo alla pila di svilupparsi verso il basso nella memoria, a causa di ripetute chiamate di funzione. Lo spazio vuoto ben visibile (o buco) che separa lo heap dal la pila è parte dello spazio degli indirizzi virtuali, ma richiede pagine fisiche realmente esistenti solo nel caso che lo heap o la pila crescano.

Ciò comporta i seguenti **vantaggi**:

- ◆ Le librerie di sistema sono condivisibili da diversi processi associando l'oggetto condiviso a uno spazio degli indirizzi virtuali, procedimento detto mappatura. Benché ciascun processo veda le librerie condivise come parte del proprio spazio degli indirizzi virtuali, le pagine che ospitano effettivamente le librerie nella memoria fisica sono in condivisione tra tutti i processi (Figura 9.3). In genere le librerie si associano allo spazio di ogni processo a loro collegato, in modalità di sola lettura.
- ◆ In maniera analoga, la memoria virtuale rende i processi in grado di condividere la memoria. Come si rammenterà dal Capitolo 3, due o più processi possono comunicare condividendo memoria. La memoria virtuale permette a un processo di creare una regione di memoria condivisibile da un altro processo. I processi che condividono questa regione la considerano parte del proprio spazio degli indirizzi virtuali, malgrado le pagine fisiche siano, in realtà, condivise, come illustrato dalla Figura 9.3.
- ◆ La memoria virtuale può consentire, per mezzo della chiamata di sistema `fopen()`, che le pagine siano condivise durante la creazione di un processo, così da velocizzare la generazione dei processi.

Paginazione su richiesta

Si consideri il caricamento in memoria di un eseguibile residente su disco. Una possibilità è quella di caricare l'intero programma nella memoria fisica al momento dell'esecuzione. Il problema, però, è che all'inizio non è detto che serva avere tutto il programma in memoria: se il programma, per esempio, fornisce all'avvio una lista di opzioni all'utente, è inutile caricare il codice per l'esecuzione di *tutte* le opzioni previste, senza tener conto di quella effettivamente scelta dall'utente. Una strategia alternativa consiste nel caricare le pagine nel momento in cui

servono realmente; si tratta di una tecnica, detta paginazione su richiesta, comunemente adottata dai sistemi con memoria virtuale. Secondo questo schema, le pagine sono caricate in memoria solo quando richieste durante l'esecuzione del programma: ne consegue che le pagine cui non si accede mai non sono mai caricate nella memoria fisica.

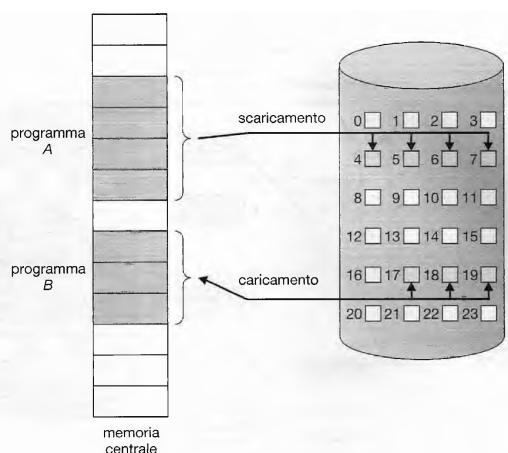


Figura 9.4 Trasferimento di una memoria paginata nello spazio contiguo di un disco.

Un sistema di paginazione su richiesta è analogo a un sistema paginato con avvicendamento dei processi in memoria; si veda la Figura 9.4. I processi risiedono in memoria secondaria, generalmente costituita di uno o più dischi. Per eseguire un processo occorre caricarlo in memoria. Tuttavia, anziché caricare in memoria l'intero processo, si può seguire un criterio d'avvicendamento "pigro" (*lazy swapping*): non si carica mai in memoria una pagina che non sia necessaria. Poiché stiamo considerando un processo come una sequenza di pagine, invece che come un unico ampio spazio d'indirizzi contiguo, l'uso del termine *avvicendamento dei processi* non è appropriato: non si manipolano interi processi ma singole pagine di processi. Nell'ambito della paginazione su richiesta, il modulo del sistema operativo che si occupa della sostituzione delle pagine si chiama paginatore (*pager*).

Concetti fondamentali

Quando un processo sta per essere caricato in memoria, il paginatore ipotizza quali pagine saranno usate, prima che il processo sia nuovamente scaricato dalla memoria. Anziché caricare in memoria tutto il processo, il paginatore trasferisce in memoria solo le pagine che ritiene necessarie. In questo modo è possibile evitare il trasferimento in memoria di pagine che non sono effettivamente usate, riducendo il tempo d'avvicendamento e la quantità di memoria fisica richiesta. Con tale schema è necessario che l'architettura disponga di un qualche meccanismo che consenta di distinguere le pagine presenti in memoria da quelle nei dischi. A tal fine è utilizzabile lo schema basato sul bit di validità. Occorre notare che indicare una pagina come non valida non sortisce alcun effetto se il processo non tenta mai di accedervi. Quindi, se l'ipotesi del paginatore è esatta e si carica no tutte e solo le pagine che servono effettivamente, il processo è eseguito proprio come se fossero state caricate tutte le pagine. Durante l'esecuzione, il processo accede alle pagine residenti in memoria, e l'esecuzione procede come di consueto. Se il processo tenta l'accesso a una pagina che non era stata caricata in memoria, l'accesso a una pagina contrassegnata come non valida causa un'eccezione di pagina mancante (*page fault trap*).

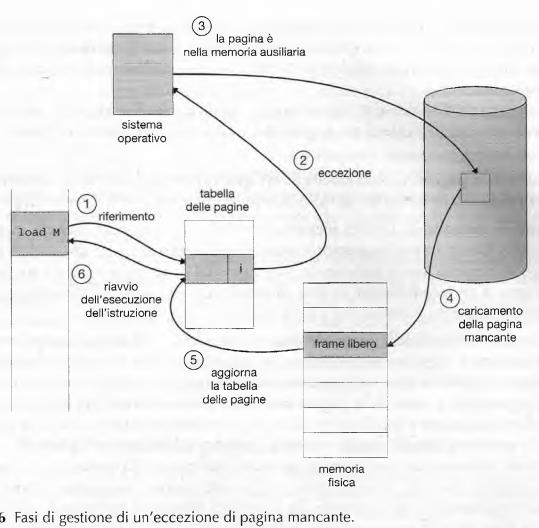


Figura 9.6 Fasi di gestione di un'eccezione di pagina mancante.

1. Si controlla una tabella interna per questo processo; in genere tale tabella è conservata insieme al blocco di controllo di processo (PCB), allo scopo di stabilire se il riferimento fosse un accesso alla memoria valido o non valido.
2. Se il riferimento non era valido, si termina il processo. Se era un riferimento valido, ma la pagina non era ancora stata portata in memoria, se ne effettua l'inserimento.
3. Si individua un frame libero, ad esempio prelevandone uno dalla lista dei frame liberi.
4. Si programma un'operazione sui dischi per trasferire la pagina desiderata nel frame appena assegnato.
5. Quando la lettura dal disco è completata, si modificano la tabella interna, conservata con il processo, e la tabella delle pagine per indicare che la pagina si trova attualmente in memoria.
6. Si riavvia l'istruzione interrotta dal segnale di eccezione. A questo punto il processo può accedere alla pagina come se questa fosse già presente in memoria.

È addirittura possibile avviare l'esecuzione di un processo *senza* pagine in memoria. Quando il sistema operativo carica nel contatore di programma l'indirizzo della prima istruzione del processo, che è in una pagina non residente in memoria, il processo accusa un'assenza di pagina. Una volta portata la pagina in memoria, il processo continua l'esecuzione, subendo assenze di pagine fino a che tutte le pagine necessarie non si trovino effettivamente in me-

moria; a questo punto si può eseguire il processo senza ulteriori richieste. Lo schema de scritto è una paginazione su richiesta pura, vale a dire che una pagina non si trasferisce in memoria finché non sia richiesta. Uno dei requisiti cruciali della paginazione su richiesta è la possibilità di rieseguire una qualunque istruzione a seguito di un'eccezione di pagina mancante o assenza di pagina (*page fault*). Avendo salvato lo stato del processo interrotto (registri, codici di condizione, contatore di programma) a causa della pagina mancante, occorrerà riavviare il processo esatta mente dallo stesso punto e con lo stesso stato, eccezion fatta per la presenza della pagina desiderata in memoria. Nella maggior parte dei casi questa situazione è piuttosto comune: un'assenza di pagina si può verificare per qualsiasi riferimento alla memoria. Se l'assenza di pagina si presenta durante la fase di prelievo di un'istruzione, l'esecuzione si può riavviare prelevando nuovamente tale istruzione. Se si verifica durante il prelievo di un operando, l'istruzione deve essere di nuovo prelevata e decodificata, quindi si può prelevare l'operando.

Come caso limite si consideri un'istruzione a tre indirizzi, come ad esempio la somma (a d d) del contenuto di A al contenuto di B, con risultato posto in C. I passi necessari per eseguire l'istruzione sono i seguenti:

- 1.prelievo e decodifica dell'istruzione (a d d);
- 2.prelievo del contenuto di A;
- 3.prelievo del contenuto di B;
- 4.addizione del contenuto di A al contenuto di B;
- 5.memorizzazione della somma in C.

Se l'assenza di pagina avviene al momento della memorizzazione in C, poiché C si trova in una pagina che non è in memoria, occorre prelevare la pagina desiderata, caricarla in memoria, correggere la tabella delle pagine e riavviare l'istruzione. Il riavvio dell'istruzione richiede una nuova operazione di prelievo della stessa, con nuova decodifica e nuovo prelievo dei due operandi; infine occorre ripetere l'addizione. In ogni modo il lavoro da ripetere non è molto, meno di un'istruzione completa, e la ripetizione è necessaria solo nel caso si verifichi un'assenza di pagina.

Prestazioni della paginazione su richiesta

La paginazione su richiesta può avere un effetto rilevante sulle prestazioni di un calcolatore. Il motivo si può comprendere calcolando il tempo d'accesso effettivo per una memoria con paginazione su richiesta. Supponendo che p sia la probabilità che si verifichi un'assenza di pagina ($0 < p < 1$), è probabile che p sia molto vicina allo zero, cioè che ci siano solo poche assenze di pagine. Il tempo d'accesso effettivo è dato dalla seguente espressione:

$$\text{tempo d'accesso effettivo} = (1-p) \times \text{tempo di gestione dell'assenza di pagina}$$

Considerando un tempo medio di servizio dell'eccezione di pagina mancante di 8 millisecondi e un tempo d'accesso alla memoria di 200 nanosecondi, il tempo effettivo d'accesso in nanosecondi è il seguente:

$$\begin{aligned} \text{tempo d'accesso effettivo} &= (1-p) \times 200 + p(8 \text{ millisecondi}) = (1-p) \times 200 + p \times 8.000.000 \\ &= 200 + 7.999.800 \times p \end{aligned}$$

Il tempo d'accesso effettivo è direttamente proporzionale alla **frequenza delle assenze di pagine** (*page-fault rate*).

Copiatura su scrittura

Si ricordi che la chiamata di sistema `fork()` crea un processo figlio come duplicato del genitore.

In alternativa, si può impiegare una tecnica nota come **copiatura su scrittura** (*copy-on-write*), il cui funzionamento si fonda sulla condivisione iniziale delle pagine da parte dei processi genitori e dei processi figli. Le pagine

condivise si contrassegnano come pagine da copiare su scrittura, a significare che, se un processo (genitore o figlio) scrive su una pagina condivisa, il sistema deve creare una copia di tale pagina. La copiatura su scrittura è illustrata rispettivamente nella Figura 9.7 e nella Figura 9.8, che mostrano il contenuto della memoria fisica prima e dopo che il processo 1 abbia modificato la pagina C.

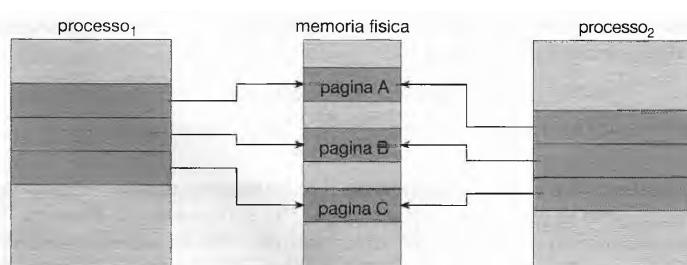


Figura 9.7 Prima della modifica alla pagina C da parte del processo₁.

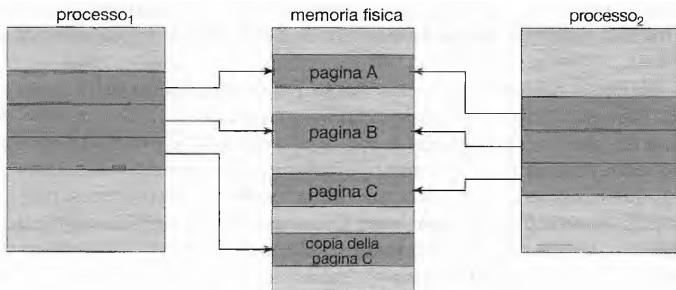


Figura 9.8 Dopo la modifica alla pagina C da parte del processo₁.

Sostituzione delle pagine

Nelle descrizioni fatte finora, la frequenza (*rate*) delle assenze di pagine non è stata un problema grave, giacché ogni pagina poteva essere assente al massimo una volta, e precisamente la prima volta in cui si effettuava un riferimento a essa. Tale rappresentazione tuttavia non è molto precisa. Se un processo di 10 pagine ne impiega effettivamente solo la metà, la paginazione su richiesta fa risparmiare l'i/O necessario per caricare le cinque pagine che non sono mai usate. Il grado di multiprogrammazione potrebbe essere aumentato eseguendo il doppio dei processi. Quindi, disponendo di 40 frame, si potrebbero eseguire otto processi anziché i quattro che si eseguirebbero se ciascuno di loro richiedesse 10 blocchi di memoria, cinque dei quali non sarebbero mai usati.

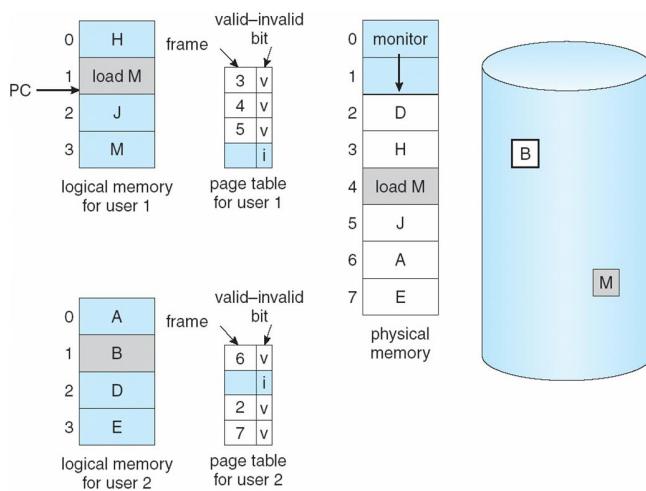
Aumentando il grado di multiprogrammazione, si **sovrassegna** la memoria. Eseguendo sei processi, ciascuno dei quali è formato da 10 pagine, di cui solo cinque sono effettivamente usate, s'incrementerebbero l'utilizzo e la produttività della CPU e si risparmierebbero 10 frame. Tuttavia è possibile che ciascuno di questi processi, per un insieme particolare di dati, abbia improvvisamente necessità di impiegare tutte le 10 pagine, perciò sarebbero necessari 60 frame, mentre ne sono disponibili solo 40.

Decidere quanta memoria assegnare all'I/O e quanta alle pagine dei programmi è un problema rilevante. Alcuni sistemi riservano una quota fissa di memoria per l'i/O, altri permettono sia ai processi utenti sia al sottosistema di i/o di competere per tutta la memoria del sistema.

La **sovrallocazione** (*over - allocation*) si può illustrare come segue. Durante l'esecuzione di un processo utente si

verifica un'assenza di pagina. Il sistema operativo determina la locazione del disco in cui risiede la pagina desiderata, ma poi scopre che la lista dei frame liberi è vuota: tutta la memoria è in uso; si veda a questo proposito la Figura 9.9.A questo punto il sistema operativo può scegliere tra diverse possibilità, ad esempio può terminare il processo utente. Tuttavia, la paginazione su richiesta è un tentativo che il sistema operativo fa per migliorare l'utilizzo e la produttività del sistema di calcolo. Gli utenti non devono sapere che i loro processi sono eseguiti su un sistema paginato. La paginazione deve essere logicamente trasparente per l'utente, quindi la terminazione del processo non costituisce la scelta migliore. Per realizzare la paginazione su richiesta è necessario risolvere due problemi principali: occorre sviluppare un algoritmo di allocazione

dei frame e un algoritmo di sostituzione delle pagine. Se sono presenti più processi in memoria, occorre decidere quanti frame vada assegnato a ciascun processo. Inoltre, quando è richiesta una sostituzione di pagina, occorre selezionare i frame da sostituire. La progettazione di algoritmi idonei a risolvere questi problemi è un compito importante, poiché l'i/O nei dischi è piuttosto oneroso. Anche miglioramenti minimi ai metodi di paginazione su richiesta ne apportano notevoli alle prestazioni del sistema.



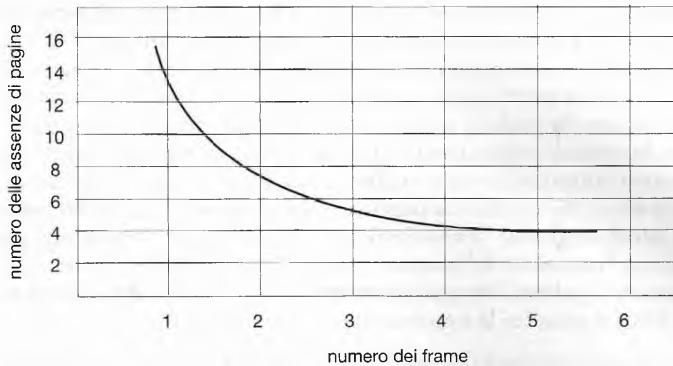
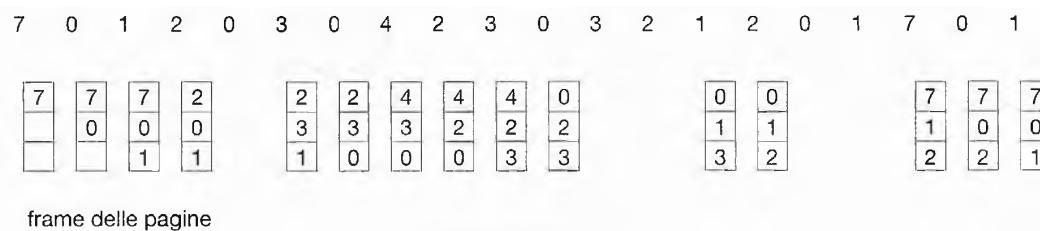


Figura 9.11 Grafico che illustra il numero di assenze di pagine rispetto al numero dei frame.

frame è necessaria una sostituzione per ogni riferimento, con il risultato di 11 assenze di pagine. In generale è prevista una curva simile a quella della Figura 9.11. Aumentando il numero dei frame, il numero di assenze di pagine diminuisce fino al livello minimo. Naturalmente aggiungendo memoria fisica il numero dei frame aumenta.

Sostituzione delle pagine secondo l'ordine d'arrivo (FIFO)

L'algoritmo di sostituzione delle pagine più semplice è un algoritmo FIFO. Questo algoritmo associa a ogni pagina l'istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire una pagina, si seleziona quella presente in memoria da più tempo. Occorre notare che non è strettamente necessario registrare l'istante in cui si carica una pagina in memoria; infatti si possono strutturare secondo una coda FIFO tutte le pagine presenti in memoria. In questo caso si sostituisce la pagina che si trova nel primo elemento della coda. Quando si carica una pagina in memoria, la si inserisce nell'ultimo elemento della coda. Questo processo prosegue come è illustrato



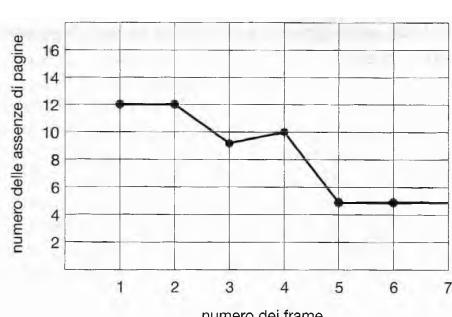
nella Figura 9.12. Le pagine presenti nei tre frame sono indicate ogni volta che si verifica un'assenza di pagina. Complessivamente si hanno 15 assenze di pagine.

Figura 9.12 Algoritmo di sostituzione delle pagine FIFO.

Problematiche se si usa FIFO:

L'algoritmo **FIFO** di sostituzione delle pagine è facile da capire e da programmare; tuttavia la sue prestazioni non sono sempre buone. La pagina sostituita potrebbe essere un modulo di inizializzazione usato molto tempo prima e che non serve più, ma potrebbe anche contenere una variabile molto usata, inizializzata precedentemente, e ancora in uso. Occorre notare che anche se si sceglie una pagina da sostituire che è in uso attivo, tutto continua a funzionare correttamente. Dopo aver rimosso una pagina attiva per inserirne una nuova, quasi immediatamente si verifica un'eccezione di pagina mancante per riprendere la pagina attiva. Per riportare la pagina attiva in memoria è necessario sostituire un'altra pagina. Quindi, scegliendo una sostituzione errata, aumenta la frequenza di pagine mancanti che rallenta l'esecuzione del processo, ma non vengono causati errori.

Nella Figura 9.13 è illustrata la curva delle assenze di pagine in funzione del numero dei frame disponibili.



Per stabilire il numero di assenze di pagine relativo a una particolare successione di riferimenti e a un particolare algoritmo di sostituzione delle pagine, occorre conoscere anche il numero dei frame disponibili. Naturalmente, aumentando il numero di quest'ultimi diminuisce il numero di assenze di pagine. Per la successione dei riferimenti precedentemente esaminata, ad esempio, dati tre o più blocchi di memoria si possono verificare tre sole assenze di pagine: una per il primo riferimento di ogni pagina. D'altra parte, se si dispone di un solo

frame è necessaria una sostituzione per ogni riferimento, con il risultato di 11 assenze di pagine. In generale è prevista una curva simile a quella della Figura 9.11. Aumentando il numero dei frame, il numero di assenze di pagine diminuisce fino al livello minimo. Naturalmente aggiungendo memoria fisica il numero dei frame aumenta.

Occorre notare che il numero delle assenze di pagine (10) per quattro frame è maggiore del numero delle assenze di pagine (9) per tre frame. Questo inatteso risultato è noto col nome di **anomalia di Belady**, e riflette il fatto che, con alcuni algoritmi di sostituzione delle pagine, la frequenza delle assenze di pagine può aumentare con l'aumentare del numero dei frame assegnati. A prima vista sembra logico supporre che fornendo più memoria a un processo le

Figura 9.13 Curva delle assenze di pagine per sostituzione FIFO su una successione di riferimenti.

prestazioni di quest'ultimo migliorino. Si è invece notato che questo presupposto non sempre è vero; l'anomalia di Belady ne è la prova.

Sostituzione ottimale delle pagine (OPT o MIN)

In seguito alla scoperta dell'anomalia di Belady, la ricerca si è diretta verso un **algoritmo ottimale di sostituzione delle pagine**. Tale algoritmo è quello che fra tutti gli algoritmi presenta la minima frequenza di assenze di pagine e non presenta mai l'anomalia di Belady. Questo algoritmo esiste ed è stato chiamato **O PT o M IN**. È

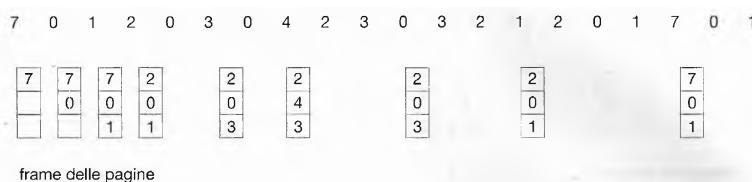


Figura 9.14 Algoritmo ottimale di sostituzione delle pagine.

conoscenza futura della successione dei riferimenti.

Sostituzione delle pagine usate meno recentemente (LRU)

Se l'algoritmo ottimale non è realizzabile, è forse possibile realizzarne un'approssimazione. La distinzione fondamentale tra gli algoritmi FIFO e OPT, oltre quella di guardare avanti o indietro nel tempo, consiste nel fatto che l'algoritmo FIFO impiega l'istante in cui una pagina è stata caricata in memoria, mentre l'algoritmo OPT impiega l'istante in cui una pagina è *usata*. Usando come approssimazione di un futuro vicino un passato recente, si sostituisce la pagina che *non è stata usata* per il periodo più lungo. Il metodo appena descritto è noto come

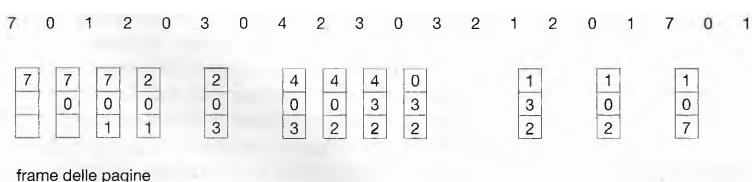


Figura 9.15 Algoritmo di sostituzione delle pagine LRU.

semplicemente: si sostituisce la pagina che non si userà per il periodo di tempo più lungo. Ad esempio, nella successione dei riferimenti considerata, l'algoritmo ottimale di sostituzione delle pagine produce nove assenze di pagine, come è mostrato nella Figura 9.14. Sfortunatamente l'algoritmo ottimale di sostituzione delle pagine è difficile da realizzare, perché richiede la

algoritmo LRU (*least recently used*). La sostituzione LRU associa a ogni pagina l'istante in cui è stata usata per l'ultima volta. Quando occorre sostituire una pagina, l'algoritmo LRU sceglie quella che non è stata usata per il periodo più lungo. Questa strategia costituisce l'algoritmo ottimale di sostituzione delle pagine con ricerca all'indietro nel tempo, anziché in avanti. Il risultato dell'applicazione dell'algoritmo LRU

alla successione dei riferimenti dell'esempio è illustrato nella Figura 9.15. L'algoritmo LRU produce 12 assenze di pagine. Nonostante questi problemi, la sostituzione LRU, con 12 assenze di pagine, è in ogni modo migliore della sostituzione FIFO, con 15 assenze di pagine.

Problemi :

Il criterio LRU si usa spesso come algoritmo di sostituzione delle pagine ed è considerato valido. Il problema principale riguarda la realizzazione della sostituzione stessa. Un algoritmo di sostituzione delle pagine LRU può richiedere una notevole assistenza da parte dell'architettura del sistema di calcolo. Il problema consiste nel determinare un ordine per i frame definito secondo il momento dell'ultimo uso. Si possono realizzare le due seguenti soluzioni:

-Contatori. Nel caso più semplice, a ogni elemento della tabella delle pagine si associa un campo del momento d'uso, e alla CPU si aggiunge un contatore che si incrementa a ogni riferimento alla memoria. Ogni volta che si fa un riferimento a una pagina, si copia il contenuto del registro contatore nel campo del momento d'uso nella tabella relativa a quella specifica pagina. In questo modo è sempre possibile conoscere il momento in cui è stato fatto l'ultimo riferimento a ogni pagina. Si sostituisce la pagina con il valore associato più piccolo.

-Pila. Un altro metodo per la realizzazione della sostituzione delle pagine LRU prevede la presenza di una pila dei numeri delle pagine. Ogni volta che si fa un riferimento a una pagina, la si estrae dalla pila e la si colloca in cima a quest'ultima. In questo modo, in cima alla pila si trova sempre la pagina usata per ultima, mentre in fondo si trova la pagina usata meno recentemente, com'è illustrato dalla Figura 9.16. Poiché alcuni elementi si devono estrarre dal centro della pila, la migliore realizzazione si ottiene usando una lista doppiamente concatenata, con un puntatore all'elemento iniziale e uno a quello finale. Per estrarre una pagina dalla pila e collocarla in cima, nel caso peggiore è

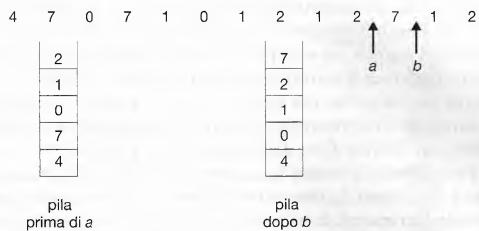


Figura 9.16 Uso di una pila per registrare i più recenti riferimenti alle pagine.

Sostituzione delle pagine per approssimazione a LRU

Sono pochi i sistemi di calcolo che dispongono di un'architettura adatta a una vera sostituzione LRU delle pagine. Nei sistemi che non

offrono tali caratteristiche specifiche si devono impiegare altri algoritmi di sostituzione delle pagine, ad esempio l'algoritmo FIFO. Molti sistemi tuttavia possono fornire un aiuto: un bit di riferimento. Il bit di riferimento a una pagina è impostato automaticamente dall'architettura del sistema ogni volta che si fa un riferimento a quella pagina, che sia una lettura o una scrittura su qualsiasi byte della pagina. I bit di riferimento sono associati a ciascun elemento della tabella delle pagine.

Inizialmente, il sistema operativo azzera tutti i bit. Quando s'inizia l'esecuzione di un processo utente, l'architettura del sistema imposta a 1 il bit associato a ciascuna pagina cui si fa riferimento. Dopo qualche tempo è possibile stabilire quali pagine sono state usate semplicemente esaminando i bit di riferimento. Non è però possibile conoscere l'*ordine* d'uso. E questa l'informazione alla base di molti algoritmi per la sostituzione delle pagine che approssimano LRU.

Algoritmo con bit supplementari di riferimento

Ulteriori informazioni sull'ordinamento si possono ottenere registrando i bit di riferimento a intervalli regolari. È possibile conservare in una tabella in memoria una serie di bit per ogni pagina. A intervalli regolari, ad esempio di 100 millisecondi, un segnale d'interruzione del timer del sistema trasferisce il controllo al sistema operativo. Questo sposta il bit di riferimento per ciascuna pagina nel bit più significativo della sequenza, traslando gli altri bit a destra di 1 bit e scartando il bit meno significativo. Questi registri a scorrimento, ad esempio di 8 bit, contengono l'ordine d'uso delle pagine relativo agli ultimi otto periodi di tempo. Se il registro a scorrimento contiene la successione di bit 00000000, significa che la pagina associata non è stata usata da otto periodi di tempo; a una pagina usata almeno una volta per ogni periodo corrisponde la successione 11111111 nel registro a scorrimento. Una pagina cui corrisponde la successione 11000100 nel relativo registro, è stata usata più recentemente di quanto non lo sia stata una cui è associata la successione 01110111. Interpretando queste successioni di bit come interi senza segno, la pagina cui è associato il numero minore è la pagina LRU, e può essere sostituita. In ogni caso l'unicità dei numeri non è garantita. Si possono sostituire (o scaricare dalla memoria all'area d'avvicendamento) tutte le pagine con il valore minore, oppure si può ricorrere a una selezione FIFO.

Il numero dei bit può ovviamente essere variato: si stabilisce secondo l'architettura disponibile per accelerarne al massimo la modifica. Nel caso limite tale numero si riduce a zero, lasciando soltanto il bit di riferimento e definendo un algoritmo noto come algoritmo di sostituzione delle pagine con seconda chance.

Algoritmo con seconda chance

L'algoritmo di base per la sostituzione con seconda chance è un algoritmo di sostituzione di tipo FIFO. Dopo aver selezionato una pagina si controlla il bit di riferimento, se il suo valore è 0, si sostituisce la pagina; se il bit di riferimento è impostato a 1, si dà una seconda chance alla pagina e la selezione passa alla successiva pagina FIFO. Quando una pagina riceve la seconda chance, si azzera il suo bit di riferimento e si aggiorna il suo istante d'arrivo al momento attuale. In questo modo, una pagina cui si offre una seconda chance non viene mai sostituita finché tutte le altre pagine non siano state sostituite, oppure non sia stata offerta loro una seconda chance. Inoltre, se una pagina è usata abbastanza spesso, in modo che il suo bit di riferimento sia sempre impostato a 1, non viene mai sostituita. Un metodo per realizzare l'algoritmo con seconda chance, detto anche a orologio (*clock*), è basato sull'uso di una coda circolare, in cui un puntatore indica qual è la prima pagina da sostituire. Quando serve un frame, si fa avanzare il puntatore finché non si trovi in corrispondenza di una pagina con il bit di riferimento 0; a ogni passo si azzera il bit di riferimento appena esaminato (Figura 9.17).

L'algoritmo con seconda chance descritto precedentemente si può migliorare considerando i bit di riferimento e di modifica (si veda il Paragrafo 9.4.1) come una coppia ordinata, con cui si possono ottenere le seguenti quattro classi: (0, 0) né recentemente usato né modificato - migliore pagina da sostituire;

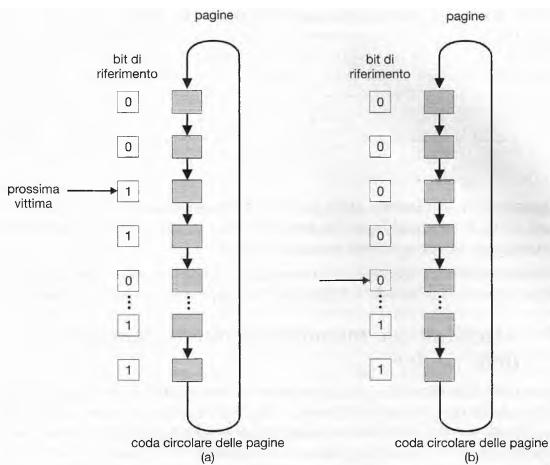


Figura 9.17 Algoritmo di sostituzione delle pagine con seconda chance (orologio).

- 1.(0, 1) non usato recentemente, ma modificato - la pagina non così buona poiché prima di essere sostituita deve essere scritta in memoria secondaria;
- 2.(1,0) usato recentemente ma non modificato —probabilmente la pagina sarà presto ancora usata;
- 3.(1,1) usato recentemente e modificato - probabilmente la pagina sarà presto ancora usata e dovrà essere scritta in memoria secondaria prima di essere sostituita.

Il numero minimo di frame per ciascun processo è definito dall’architettura, mentre il numero massimo è definito dalla quantità di memoria fisica disponibile. Rimane ancora aperta la questione della scelta dell’allocazione dei frame.

Algoritmi di allocazione

Il modo più semplice per suddividere m frame tra n processi è quello per cui a ciascuno si dà una parte uguale, min frame. Dati 93 frame e cinque processi, ogni processo riceve 18 fra me. I tre frame lasciati liberi si potrebbero usare come gruppo dei frame liberi. Questo schema è chiamato **allocazione uniforme**.

Un’alternativa consiste nel riconoscere che diversi processi hanno bisogno di quantità di memoria diverse. Si consideri un sistema con frame di 1 KB. Se un piccolo processo utente di 10 KB e una base di dati interattiva di 127 KB sono gli unici due processi in esecuzione su un sistema con 62 frame liberi, non ha senso allocare a ciascun processo 31 frame. Al pro cesso utente non ne servono più di 10, quindi gli altri 21 sarebbero semplicemente sprecati. Per risolvere questo problema è possibile ricorrere all’**allocazione proporzionale**, secondo cui la memoria disponibile si assegna a ciascun processo secondo la propria dimensione.

Occorre notare che sia con l’allocazione uniforme sia con l’allocazione proporzionale, un processo a priorità elevata è trattato come un processo a bassa priorità anche se, per definizione, si vorrebbe che al processo con elevata priorità fosse allocata più memoria per accelerarne l’esecuzione, a discapito dei processi a bassa priorità. Un soluzione prevede l’uso di uno schema di allocazione proporzionale in cui il rapporto dei frame non dipende dalle dimensioni relative dei processi, ma dalle priorità degli stessi oppure da una combinazione di dimensioni e priorità.

Allocazione globale e allocazione locale

Un altro importante fattore che riguarda il modo in cui si assegnano i frame ai vari processi è la sostituzione delle pagine. Nei casi in cui vi siano più processi in competizione per i frame, gli algoritmi di sostituzione delle pagine si possono classificare in due categorie generali: **sostituzione globale** e **sostituzione locale**. La sostituzione globale permette che per un processo si scelga un frame per la sostituzione dall’insieme di tutti i frame, anche se quel frame è correntemente allocato a un altro processo; un processo può dunque sottrarre un frame a un altro processo. La sostituzione locale richiede invece che per ogni processo si scelga un frame solo dal proprio insieme di frame.

Si consideri ad esempio uno schema di allocazione che, per una sostituzione a favore dei processi ad alta priorità, permetta di sottrarre frame ai processi a bassa priorità. Per un pro cesso si può stabilire una sostituzione che attinga tra i suoi frame oppure tra quelli di qualsiasi processo con priorità minore. Questo metodo permette a un processo ad alta priorità di aumentare il proprio livello di allocazione dei frame a discapito del processo a bassa priorità.

Con la strategia di sostituzione locale, il numero di blocchi di memoria assegnati a un processo non cambia.

Con la sostituzione globale, invece, può accadere che per un certo pro cesso si selezionino solo frame allocati ad altri processi, aumentando così il numero di frame assegnati a quel processo, purché per altri non si scelgano per la sostituzione i *propri* frame.

L’algoritmo di sostituzione globale risente di un problema: un processo non può controllare la propria frequenza di assenze di pagine (*page-fault rate*), infatti l’insieme di pagine che si trova in memoria per un processo non dipende solo dal comportamento di paginazione di quel processo, ma anche dal comportamento di paginazione di altri processi.

Con l’algoritmo di sostituzione locale questo problema non si presenta. Infatti l’insieme di pagine in memoria per un processo subisce l’effetto del comportamento di paginazione di quel solo processo. Dal canto suo, **la sostituzione locale può limitare un processo, non rendendogli disponibili altre pagine di memoria meno usate. Generalmente, la sostituzione globale genera una maggiore produttività del sistema, e perciò è il metodo più usato.**

Paginazione degenera (thrashing)

Se il numero dei frame allocati a un processo con priorità bassa diviene inferiore al numero minimo richiesto dall'architettura del calcolatore, occorre sospendere l'esecuzione del processo, e quindi togliere le pagine restanti, liberando tutti i frame allocati. Questa operazione introduce un livello intermedio di scheduling per la gestione dell'entrata e dell'uscita dei processi in memoria centrale.

Infatti, si consideri un qualsiasi processo che non disponga di un numero di frame "sufficiente". Anche se tecnicamente si può ridurre al valore minimo il numero dei frame allocati, esiste un certo (in generale grande) numero di pagine in uso attivo. Se non dispone di questo numero di frame, il processo accusa immediatamente un'assenza di pagina. A questo punto si deve sostituire qualche pagina; ma, poiché tutte le sue pagine sono in uso attivo, si deve sostituire una pagina che sarà immediatamente necessaria, e di conseguenza si verificano subito parecchie assenze di pagine. Il processo continua a subire assenze di pagine, facendo sostituire pagine che saranno immediatamente trattate come assenti e dovranno essere riprese. Questa intensa quanto degenera paginazione (nota come *thrashing*) si verifica quando si spende più tempo per la paginazione che per l'esecuzione dei processi.

-Cause della paginazione degenera :

La degenerazione dell'attività di paginazione causa parecchi problemi di prestazioni. Si consideri il seguente scenario, basato sul comportamento effettivo dei primi sistemi di paginazione. Il sistema operativo vigila sull'utilizzo della CPU. Se questo è basso, aumenta il grado di multiprogrammazione introducendo un nuovo processo. Si usa un algoritmo di sostituzione delle pagine globale, che sostituisce le pagine senza tener conto del processo al quale appartengono. Per ora si ipotizzi che un processo entri in una nuova fase d'esecuzione e richieda più fra me; se ciò si verifica si ha una serie di assenze di pagine, cui segue la sottrazione di nuove pagine ad altri processi. Questi processi hanno però bisogno di quelle pagine e quindi subiscono anch'essi delle assenze di pagine, con conseguente sottrazione di pagine ad altri processi. Per effettuare il caricamento e lo scaricamento delle pagine per questi processi si deve usare il dispositivo di paginazione. Mentre si mettono i processi in coda per il dispositivo di paginazione, la coda dei processi pronti per l'esecuzione si svuota, quindi l'utilizzo della CPU diminuisce.

Lo scheduler della CPU rileva questa riduzione dell'utilizzo della CPU e *aumenta* il grado di multiprogrammazione. Si tenta di avviare il nuovo processo sottraendo pagine ai processi in esecuzione, causando ulteriori assenze di pagine e allungando la coda per il dispositivo di paginazione. L'utilizzo della CPU scende ulteriormente, e lo scheduler della CPU tenta di aumentare ancora il grado di multiprogrammazione. L'attività di paginazione è degenerata in una situazione patologica che fa precipitare la produttività del sistema. La frequenza delle assenze di pagine aumenta in modo impressionante, e di conseguenza aumenta il tempo effettivo d'accesso alla memoria. I processi non svolgono alcun lavoro, poiché si sta spendendo tutto il tempo nell'attività di paginazione.

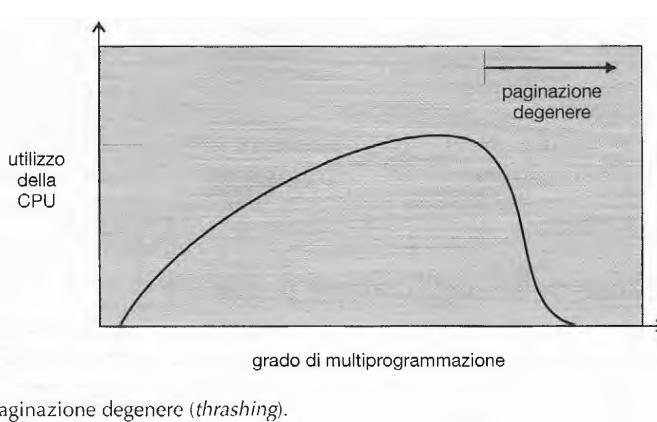


Figura 9.18 Paginazione degenera (*thrashing*).

Questo fenomeno è illustrato nella Figura 9.18, in cui si riporta l'utilizzo della CPU in funzione del grado di multiprogrammazione. In questa situazione, per aumentare l'utilizzo della CPU occorre *ridurre* il grado di multiprogrammazione. Gli effetti di questa situazione si possono limitare usando un algoritmo di sostituzione locale, o algoritmo di sostituzione per priorità. Con la sostituzione locale, se un processo ricade nell'attività di paginazione degenera, non può sottrarre frame a un altro processo e quindi provocarne a sua volta la degenerazione. Le pagine si sostituiscono tenendo conto del processo di cui fanno parte.

Tuttavia, se i processi la cui attività di paginazione degenera rimangono nella coda d'attesa del dispositivo di paginazione per la maggior parte del tempo. Il tempo di servizio medio di un'eccezione di pagina mancante aumenta a causa dell'allungamento medio della coda d'attesa del dispositivo di paginazione. Di conseguenza, il tempo effettivo d'accesso al dispositivo di paginazione aumenta anche per gli altri processi. Per evitare il verificarsi di queste situazioni, occorre fornire a un processo tutti i frame di cui necessita. Per cercare di sapere quanti frame "servano" a un processo si impiegano diverse tecniche. Il modello dell'insieme di lavoro (**working-set**), trattato nel Paragrafo 9.6.2, comincia osservando quanti siano i frame che un processo sta effettivamente usando. Questo

metodo definisce il modello di località d'esecuzione del processo. Il modello di località stabilisce che un processo, durante la sua esecuzione, si sposta di località in località.

Frequenza delle assenze di pagine

Il modello dell'insieme di lavoro riscuote un discreto successo, e la sua conoscenza può servire per la prepaginazione (Paragrafo 9.9.1), ma appare un modo alquanto goffo per con trollare la degenerazione della paginazione. La strategia basata sulla frequenza delle assenze di pagine (**page fault frequency**, PFF) è più diretta. Il problema specifico è la prevenzione della paginazione degenere. La frequenza delle assenze di pagine in tale situazione è alta, ed è proprio questa che si deve controllare. Se la frequenza delle assenze di pagine è eccessiva, significa che il processo necessita di più frame. Analogamente, se la frequenza delle assenze di pagine è molto bassa, il processo potrebbe disporre di troppi frame. Si può fissare un limite inferiore e un limite superiore per la

frequenza desiderata delle assenze di pagine, com'è illustrato nella Figura 9.21. Se la frequenza effettiva delle assenze di pagine per un processo oltrepassa il limite superiore, occorre allocare a quel processo un altro frame; se la frequenza scende sotto il limite inferiore, si sottrae un frame a quel processo. Quindi, per prevenire la paginazione degenere, si può misurare e controllare direttamente la frequenza delle assenze di pagine.

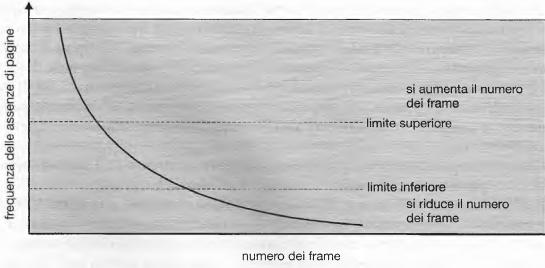


Figura 9.21 Frequenza delle assenze di pagine.

Sommario Capitolo Nove

E auspicabile poter eseguire processi il cui spazio degli indirizzi logici superi quello disponibile per gli indirizzi fisici. La memoria virtuale è una tecnica che permette di associare grandi spazi degli indirizzi logici a quantità più ridotte di memoria fisica. La memoria virtuale è una tecnica che consente di eseguire processi molto grandi e di aumentare il grado di multiprogrammazione, incrementando l'utilizzo della CPU. Inoltre, grazie a tale tecnica, i programmatore di applicazioni non devono più preoccuparsi della disponibilità di memoria. In più, grazie alla memoria virtuale, processi distinti possono condividere librerie di sistema e memoria. La memoria virtuale consente anche l'utilizzo di un tipo efficiente di creazione di processo conosciuto con il nome di copiatura su scrittura (*copy-on-write*), in cui i processi genitore e figlio condividono pagine effettive di memoria.

La memoria virtuale è comunemente implementata tramite paginazione su richiesta, che trasferisce in memoria una pagina solo quando si incontra un riferimento alla pagina stessa; il primo riferimento produce un errore di pagina. Il kernel del sistema operativo consulta una tabella interna per stabilire la locazione della pagina in memoria ausiliaria, quindi individua un frame libero e vi trasferisce la pagina prelevandola dalla memoria ausiliaria. La tabella delle pagine viene aggiornata per riflettere tale modifica e si riavvia l'istruzione che aveva causato l'eccezione di pagina mancante. Questo metodo permette l'esecuzione di un processo anche se in memoria centrale non è interamente presente la sua immagine di memoria. Finché la frequenza delle assenze di pagine rimane ragionevolmente bassa, le prestazioni si considerano accettabili.

La paginazione su richiesta si può usare per ridurre il numero dei frame assegnati a un processo. Questo metodo può aumentare il grado di multiprogrammazione, permettendo che più processi siano disponibili per l'esecuzione in un dato momento e, almeno in teoria, può migliorare l'utilizzo della CPU. Inoltre, consente l'esecuzione di processi i cui requisiti di spazio di memoria superano la memoria fisica disponibile. Tali processi si eseguono in memoria virtuale.

Se i requisiti di spazio di memoria superano la memoria fisica, può essere necessaria la sostituzione di pagine presenti in memoria allo scopo di liberare frame per nuove pagine. Gli algoritmi usati per la sostituzione delle pagine sono diversi: la sostituzione di tipo FIFO è facile da programmare, ma soffre dell'anomalia di Belady; la sostituzione ottimale delle pagine richiede la conoscenza dei futuri riferimenti alla memoria; la sostituzione delle pagine LRU è quasi ottimale, ma può essere di difficile realizzazione. Quasi tutti gli algoritmi di sostituzione delle pagine, come l'algoritmo con seconda chance, sono approssimazioni della sostituzione LRU.

Oltre un algoritmo di sostituzione delle pagine, occorre un criterio di allocazione dei frame. L'allocazione può essere statica, indicando una sostituzione di pagine locale, oppure dinamica, con una sostituzione di pagine globale.

Il modello dell'insieme di lavoro presuppone che i processi siano eseguiti in località. L'insieme di lavoro è l'insieme delle pagine nel la località corrente. Di conseguenza, a ogni processo si possono allocare frame sufficienti al suo corrente insieme di lavoro. Se un processo non ha spazio di memoria sufficiente per il proprio insieme di lavoro, si ha una paginazione degenere (*thrashing*). Se a ogni processo si devono fornire frame sufficienti per evitare tale degenerazione, sono necessarie le attività d'avvicendamento (*swapping*) e scheduling dei processi. La maggior parte dei sistemi operativi mette a disposizione degli strumenti per la mappatura in memoria dei file, ciò che permette di trattare l'i/O alla stregua degli accessi in memoria. La API Win32 implementa la condivisione della memoria tramite la mappatura in memoria di file.

Di solito, i processi del kernel richiedono l'allocazione di pagine fisicamente contigue. Il sistema buddy alloca memoria al kernel in segmenti di dimensioni pari a potenze di 2; ciò conduce facilmente a frammentazione. L'allocazione a lastre assegna le strutture dati del kernel a cache associate a lastre, le quali a loro volta sono costituite da una o più pagine fisi che contigue. Questa strategia non produce frammentazione e permette di servire rapidamente le richieste del kernel.

La corretta progettazione dei sistemi di paginazione non solo richiede la soluzione dei due problemi fondamentali della sostituzione delle pagine e dell'allocazione dei frame, ma porta anche a considerare questioni relative a dimensione delle pagine, i/o, gestione dei lock, prepaginazione, generazione dei processi, struttura dei programmi, e altro ancora.

Interfaccia del file system

Per la maggior parte degli utenti il file system è l'aspetto più visibile di un sistema operativo. Esso fornisce il meccanismo per la registrazione e l'accesso in linea a dati e programmi appartenenti al sistema operativo e a tutti gli utenti del sistema di calcolo.

Il file system consiste di due parti distinte: un insieme di **file**, ciascuno dei quali contenente dati correlati, e una **struttura della directory**, che organizza tutti i file nel sistema e fornisce le informazioni relative.

Un file è un insieme di informazioni, correlate e registrate in memoria secondaria, cui è stato assegnato un nome. Dal punto di vista dell'utente, un file è la più piccola porzione di memoria secondaria logica; i dati si possono cioè scrivere in memoria secondaria soltanto all'interno di un file. Di solito i file rappresentano programmi, in forma sorgente e oggetto, e dati. I file di dati possono essere numerici, alfabetici, alfanumerici o binari, e non possedere un formato specifico, come i file di testo; oppure essere rigidamente formattati. In genere un file è formato da una sequenza di bit, byte, righe o *record* il cui significato è definito dal creatore e dall'utente del file stesso. Il concetto di file è quindi estremamente generale. Un file ha una **struttura** definita secondo il tipo: un file di *testo* è formato da una sequenza di caratteri organizzati in righe, e probabilmente pagine; un file *sorgente* è formato da una sequenza di procedure e funzioni, ciascuna delle quali è a sua volta organizzata in dichiarazioni seguite da istruzioni eseguibili; un file *oggetto* è formato da una sequenza di byte, organizzati in blocchi, comprensibile al modulo di collegamento del sistema; un file *eseguibile* consiste di una serie di sezioni di codice che il caricatore può caricare in memoria ed eseguire.

—>Attributi dei file :Un file ha altri attributi che possono variare secondo il sistema operativo, ma che tipicamente comprendono i seguenti.

Nome. Il nome simbolico del file è l'unica informazione in forma umanamente leggibile.

Identificatore. Si tratta di un'etichetta unica, di solito un numero, che identifica il file all'interno del file system; è il nome impiegato dal sistema per il file.

Tipo. Questa informazione è necessaria ai sistemi che gestiscono tipi di file diversi.

Locazione. Si tratta di un puntatore al dispositivo e alla locazione del file in tale dispositivo.

Dimensione. Si tratta della dimensione corrente del file (in byte, parole o blocchi) ed eventualmente della massima dimensione consentita.

Protezione. Le informazioni di controllo degli accessi controllano chi può leggere, scrivere o far eseguire il file.

Ora, data e identificazione dell'utente. Queste informazioni possono essere relative alla creazione, l'ultima modifica e l'ultimo uso. Questi dati possono essere utili ai fini della protezione e per controllarne l'uso.

Le informazioni sui file sono conservate nella struttura della directory, che risiede a sua volta in memoria secondaria.

—> Operazioni sui file :

Creazione di un file.

Scrittura di un file.

Lettura di un file

Riposizionamento in un file.

Cancellazione di un file.

Troncamento di un file.

Metodi d'accesso

I file memorizzano informazioni; al momento dell'uso è necessario accedere a queste informazioni e trasferirle in memoria.

Accesso sequenziale

Il più semplice metodo d'accesso è l'accesso sequenziale: le informazioni del file si elaborano ordinatamente, un record dopo l'altro; questo metodo d'accesso è di gran lunga il più comune. Le più comuni operazioni che si compiono sui file sono le letture e le scritture: un'operazione di lettura legge la prima porzione e fa avanzare automaticamente il puntatore del file che tiene traccia della locazione di I/O; analogamente, un'operazione di scrittura fa un'aggiunta in coda al file e avanza fino alla fine delle informazioni appena scritte, che costituisce la nuova fine del file. Un file siffatto si può reimpostare sull'inizio e, in alcuni sistemi, un programma può riuscire ad andare avanti o indietro di n record, con n intero e alcune volte solo per $n - 1$.

L'accesso sequenziale è illustrato nella Figura 10.3. L'accesso sequenziale è basato su un modello di file che si rifa al nastro, e funziona nei dispositivi ad accesso sequenziale così come nei dispositivi ad accesso diretto.

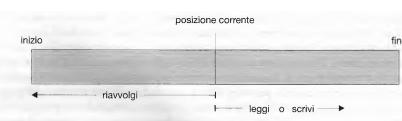


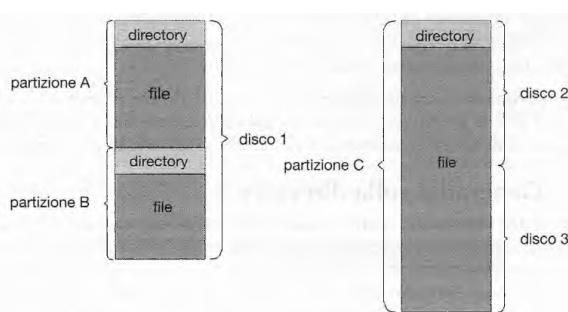
Figura 10.3 File ad accesso sequenziale.

Accesso diretto

Un altro metodo è l'accesso diretto (o accesso relativo). Un file è formato da elementi logici (record) di lunghezza fissa; ciò consente ai programmi di leggere e scrivere rapidamente tali elementi senza un ordine particolare. Il metodo ad accesso diretto si fonda su un modello di file che si rifa al disco: i dischi permettono, infatti, l'accesso diretto a ogni blocco di file. Il file si considera come una sequenza numerata di blocchi o record che si possono leggere o scrivere in modo arbitrario: si può ad esempio leggere il blocco 14, quindi il blocco 53 e poi scrivere il blocco 7. Non esistono limiti all'ordine di lettura o scrittura di un file ad accesso diretto.

I file ad accesso diretto sono molto utili quando è necessario accedere immediatamente a grandi quantità di informazioni. Spesso le basi di dati sono di questo tipo: quando si presenta un'interrogazione riguardante un oggetto particolare, occorre stabilire quale blocco contiene la risposta alla richiesta e quindi leggere direttamente quel blocco, ottenendo così le informazioni richieste.

Struttura della directory e del disco



Ogni volume contenente un file system deve anche avere in sé le informazioni sui file presenti nel sistema. Tali informazioni risiedono in una **directory del dispositivo** o **indice del volume**. La directory del dispositivo (in breve **directory**) registra informazioni, quali nome, posizione e tipo, di tutti i file del volume. La Figura 10.6 mostra la tipica organizzazione dei file system.

Figura 10.6 Tipica organizzazione di un file system.

Generalità sulla directory

La directory si può considerare come una tabella di simboli che traduce i nomi dei file negli elementi in essa contenuti. Da questo punto di vista, si capisce che la stessa directory si può organizzare in molti modi diversi; deve essere possibile inserire nuovi elementi, cancellarne esistenti, cercare un elemento ed elencare tutti gli elementi della directory.

Ricerca di un file. Deve esserci la possibilità di scorrere una directory per individuare l'elemento associato a un particolare file. Poiché i file possono avere nomi simbolici, e poiché nomi simili possono indicare relazioni tra file, deve esistere la possibilità di trovare tutti i file il cui nome soddisfi una particolare espressione.

Creazione di un file. Deve essere possibile creare nuovi file e aggiungerli alla directory.

Cancellazione di un file. Quando non serve più, si deve poter rimuovere un file dalla directory.

Elencazione di una directory. Deve esistere la possibilità di elencare tutti i file di una directory, e il contenuto degli elementi della directory associati ai rispettivi file nell'elenco.

Ridenominazione di un file. Poiché il nome di un file rappresenta per i suoi utenti il contenuto del file, questo nome deve poter essere modificato quando il contenuto o l'uso del file subiscono cambiamenti. La ridenominazione di un file potrebbe comportare la variazione della posizione del file nella directory.

Attraversamento del file system.

Metodi di allocazione

La natura ad accesso diretto dei dischi permette una certa flessibilità nella realizzazione dei file. In quasi tutti i casi, molti file si memorizzano nello stesso disco. Il problema principale consiste dunque nell'allocare lo spazio per questi file in modo che lo spazio nel disco sia usato efficientemente e l'accesso ai file sia rapido. Esistono tre metodi principali per l'allocazione dello spazio di un disco; può essere infatti **contigua**, **concatenata** o **indicizzata**. Ciascuno di questi metodi presenta vantaggi e svantaggi.

Allocazione contigua

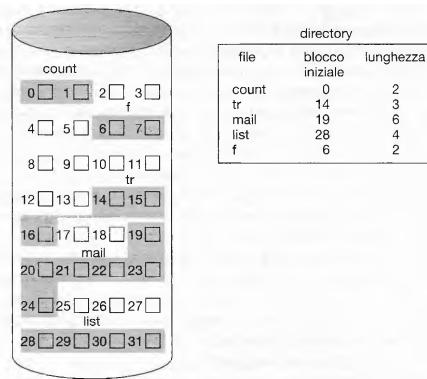


Figura 11.5 Allocazione contigua dello spazio dei dischi.

Per usare il metodo di **allocazione contigua**, ogni file deve occupare un insieme di blocchi contigui del disco. L'allocazione contigua dello spazio per un file è definita dall'indirizzo del primo blocco (inteso come numero di blocco) e dalla lunghezza (espressa in blocchi). Se il file è lungo n blocchi e comincia dalla locazione b , allora occupa i blocchi $b, b + 1, b + 2, \dots, b + n - 1$. L'elemento di directory per ciascun file indica l'indirizzo del blocco d'inizio e la lunghezza dell'area assegnata per questo file (Figura 11.5). Accedere a un file il cui spazio è assegnato in modo contiguo è facile. Quando si usa un accesso sequenziale, il file system memorizza l'indirizzo dell'ultimo blocco cui è stato fatto riferimento e, se è necessario, legge il blocco successivo. Nel caso di un accesso diretto al blocco i di un file che comincia al blocco b si può

accedere immediatamente al blocco $b + i$. Quindi, sia l'accesso sequenziale sia quello diretto si possono gestire con l'allocazione contigua. L'allocazione contigua presenta però alcuni problemi; una difficoltà riguarda l'individuazione dello spazio per un nuovo file. Il problema dell'allocazione contigua dello spazio dei dischi si può considerare un'applicazione particolare del problema generale dell'allocazione dinamica della memoria, trattato nel Paragrafo 8.3; il problema generale è, infatti, quello di soddisfare una richiesta di dimensione n data una lista di buchi liberi. I più comuni criteri di scelta di un buco libero da un insieme di buchi disponibili sono quelli del primo buco abbastanza grande (*first-fit*) e del più piccolo tra i buchi abbastanza grandi (*best-fit*). Simulazioni hanno dimostrato che questi due criteri sono più efficienti di quello di scelta del buco più grande (*worst-fit*) sia in termini di tempo sia d'uso della memoria. Questi algoritmi soffrono della frammentazione esterna: assegnando e liberando lo spazio per i file, lo spazio libero dei dischi viene frammentato in tanti piccoli pezzi. La frammentazione esterna si ha ogniqualvolta lo spazio libero è suddiviso in pezzi, e diviene un problema quando il più grande di tali pezzi

contigui non è sufficiente a soddisfare una richiesta; la memoria viene frammentata in tanti buchi, nessuno dei quali è abbastanza grande da contenere i dati. Questo schema compatta efficacemente tutto lo spazio libero in uno spazio contiguo, risolvendo il problema della frammentazione. Il costo di questa compattazione è rappresentato dal tempo necessario, ed è particolarmente pesante per i dischi di grande capacità che impiegano l'allocazione contigua.

Allocazione concatenata

L'allocazione concatenata risolve tutti i problemi dell'allocazione contigua. Con questo tipo di allocazione, infatti, ogni file è composto da una lista concatenata di blocchi del disco i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo e all'ultimo blocco del file. Ad esempio, un file di cinque blocchi può cominciare dal blocco 9, continuare al blocco 16, quindi al blocco 1, al blocco 10 e infine terminare al blocco 25 (Figura 11.6). Ogni blocco contiene un puntatore al blocco successivo. Questi puntatori non sono disponibili all'utente, quindi se ogni blocco è formato di 512 byte e un indirizzo del disco (il puntatore) richiede 4 byte, l'utente vede blocchi di 508 byte.

